AD-A050 966   LOGICON INC SAN DIEGO CALIF   F/G 5/2

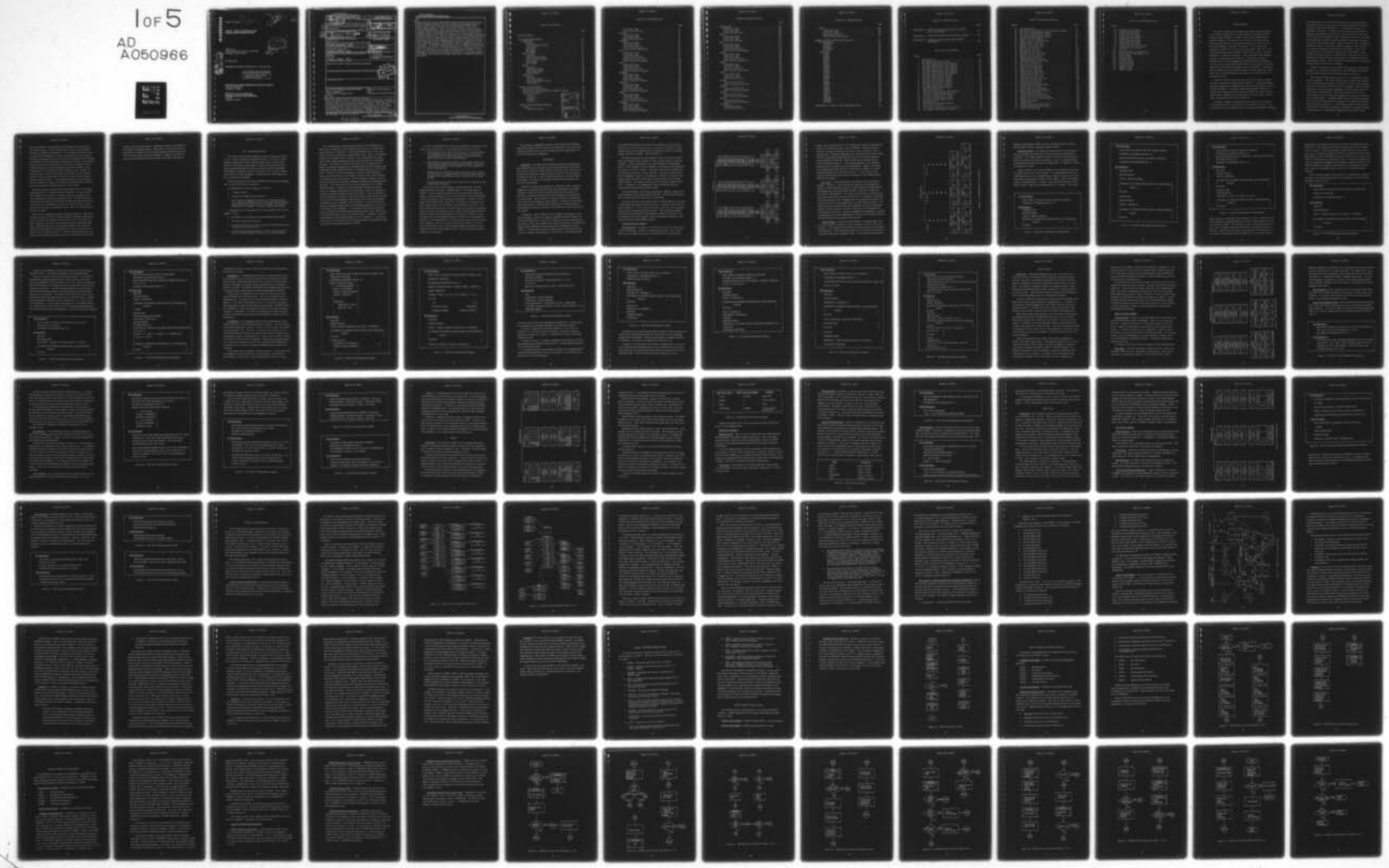
ADAPT I FINAL FUNCTIONAL AND SYSTEM DESIGN SPECIFICATION.(U)

JAN 78 L R ERICKSON, M E SOLEGLAD   N00014-76-C-0899

UNCLASSIFIED   76-C-0899-2-F   NL

1 OF 5

AD
A050966



12

Report 76-C-0899-2

# ADAPT I FINAL FUNCTIONAL AND SYSTEM DESIGN SPECIFICATION

Logicon, Inc.
4010 Sorrento Valley Blvd., P.O. Box 80158
San Diego, California 92138

30 January 1978

Final Report for Period 1 November 1977 - 30 January 1978

| DOD DISTRIBUTION STATEMENT |
|---|
| Approved for public release; distribution unlimited. |

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
1400 Wilson Boulevard
Arlington, VA 22209

OFFICE OF NAVAL RESEARCH
INFORMATION SYSTEMS PROGRAM
Code 437
Arlington, VA 22217

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| | |
|---|---|
| 14 **REPORT DOCUMENTATION PAGE** | READ INSTRUCTIONS BEFORE COMPLETING FORM |
| 1. REPORT NUMBER 76-C-Ø899-2-F / 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER 9 |
| 4. TITLE (and Subtitle) 6 ADAPT I Final Functional and System Design Specification. | 5. TYPE OF REPORT & PERIOD COVERED Final Rep[redacted]t, 1 Nov [redacted]77-30 Jan [redacted]78 |
| | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s) 10 L. R. Erickson, M. E. Soleglad [redacted] S. L. Westermark | 8. CONTRACT OR GRANT NUMBER(s) 15 NØØØ14-76-C-Ø899 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS Logicon, Inc. 4010 Sorrento Valley Boulevard, P.O. Box 80158 San Diego, California 92138 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, Virginia 22209 | 11 12. REPORT DATE 30 Jan[redacted]78 |
| | 13. NUMBER OF PAGES 12 426 p. |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Program Code 437 Arlington, Virginia 22217 | 15. SECURITY CLASS. (of this report) Unclassified |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Data Base Management Systems (DBMS)
Query Languages
Uniform Data Language (UDL)
Network
Language Transformations
UNIX
Terminal Access System (TAS)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Under the ADAPT project, a prototype intelligent terminal will be developed which provides users and/or other systems a uniform interface for accessing multiple online databases located on different systems. The underlying technology applied by ADAPT will be the transformation from one uniform data language, UDL, to other target query languages which reside on a network. The four database systems/query languages will be transformed to are the SIGINT On-Line Information System (SOLIS), Defense Intelligence

→ next page

406 012

All

20.

Agency On-Line System (DIAOLS)/Intelligence Support System (ISS), Data Base Management System 1100 (DMS-1100)/Query Language Processor (QLP), and Technical Information Processing Systems (TIPS)/TIPS Interrogation Language (TILE). The ADAPT system is comprised of 23 distinct UNIX processes. Significant processes include the Data Definition Language (DDL) process, which provides ADAPT I users with a full file definition facility via an interactive language; the Transformation Definition Language (TDL) process, which provides a means of specifying transformation specific data to ADAPT I; two language transformation processes (one for true interactive query languages and the other for transaction-oriented (batch) database systems); and a database response translation processer which maps all distant host data responses onto a compatible UDL data representation. All ADAPT I processes operate over a UNIX hierarchical file structure composed of a set of UNIX directories and internally structured files. The structured files comprise the global data for ADAPT I. Key to the language transformations performed by ADAPT I is the normalization of selection criteria (generalized boolean expressions) to disjunctive-normal-form (DNF).

## TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

## TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Cont)

## LIST OF ILLUSTRATIONS (Cont)

## INTRODUCTION

This report constitutes a functional and system design specification for the ARPA Data Base Access and Presentation Terminal (ADAPT) system, phase I (ADAPT I). ADAPT, which will be implemented in a sequence of phases, will provide network users with a uniform data base accessing interface. This interface will be in the form of a data base query language comprised of two parts: a set of logical data structures in which a user can view all network files, and a set of user-oriented statements and commands by which users can query, display, manipulate, or update network files. These statements and commands operate over the logical data structures. ADAPT also provides to privileged users two sub-languages for defining network-resident files in terms of the aforementioned logical data structures and for specifying distant host file-specific information required by ADAPT. The primary goal of ADAPT is to provide network users a uniform interface for data base access, thus eliminating the learning curve associated with each dissimilar system.

Initially, ADAPT only provides general file interrogation capabilities plus a limited data display facility (ADAPT I). In additional phases, full file maintenance and sophisticated report generation facilities will be provided (ADAPT II), complete programming language constructs for dynamic file manipulation will be provided (ADAPT III), and a local data base manager capability will be added which will allow users to create, query, and maintain files locally (ADAPT IV).

Essential to ADAPT development is the existence of the Terminal Access System (TAS), now operational on the COINS II network. TAS provides the crucial network interfaces required by the ADAPT technology.

This includes the nontrivial COINS I batch protocol still operational on the COINS I network as well as more conventional interactive interfaces commonly found on the ARPANET (i. e. , the COINS II network is of ARPANET technology). Currently, two of the systems which ADAPT I addresses are assumed to be batch in nature and therefore require this COINS I protocol as emulated on TAS. Analogously, the other two target hosts addressed by ADAPT I require interactive connections and transparent user logon to these hosts, a facility provided by TAS. All network transactions, batch or interactive, must be logged as COINI records for post statistical analysis procedures performed offline. Again, TAS provides this required facility for all its network operations. Since ADAPT I allows users access to COINS II files, the proper user access authorization checks must be performed prior to initiating queries and perusing query results. TAS, which has the same security requirement, again provides this interface for ADAPT I.

ADAPT I constitutes the first COINS II user application system that operates as a true subsystem under TAS. All TAS/ADAPT I interfaces are described in detail in the body of this design specification.

The remainder of this document consists of four additional sections and four appendices. The next section provides a functional description of the transformations required to transform the Uniform Data Language (UDL) into four distant host data base query languages. These data base management systems and query languages are DIAOLS/ISS, DMS-1100/QLP, SOLIS, and TIPS/TILE. This section shows how UDL can be transformed to these four data base management system query languages in a consistent manner. Although ADAPT I deals only with transformations involving file interrogations and display, the complete transformation set has been provided; i. e. , interrogation, display, and file update. Since ADAPT II will provide additional capabilities, including file update transformations, it is important to demonstrate that the UDL subset provided in ADAPT I can be expanded (not changed) to incorporate the new transformations. The third section gives a general description of the ADAPT I operational environment. This includes a detailed description of the file environment over which the ADAPT I processes operate. All UNIX file pathnames are described and the mechanisms and/or naming conventions for constructing dynamic pathnames are given. Also included in this section is a brief description of the interactive ADAPT I System Generation programs required to generate the initial ADAPT I file environment and to register and/or delete ADAPT I users. Finally, this section provides a general description of the overall data flow between ADAPT I processes with respect to network responses and statement/command entries by users. It is important that this section be read before one attempts to read the more detailed descriptions of the ADAPT I processes and global data files provided in later sections.

The fourth section provides detailed descriptions of the 15 processes comprising ADAPT I. Included in these descriptions are lists of global data utilization, descriptions of important functions, and general flow charts illustrating the data flow of each process. The fifth section provides detailed descriptions of the ADAPT I global functions and data files. The ADAPT I file environment discussion provided in the third section should be studied before reading this section, since it ties all global data structures together.

There are four appendices included in this specification. Appendix A provides the actual YACC (Yet Another Compiler-Compiler) input specifications (grammer descriptions) for the ADAPT I UDL, DDL (Data Defination Language), and TDL (Transformation Definition Language) languages. Appendix B provides a very detailed example of the interaction between a file dictionary for a hypothetical UDL file and sample record data as they would exist in ADAPT I. Before reading this appendix, one should study the applicable global data structure descriptions in the fifth section of the basic specification. Appendix C contains a description of the ADAPT I decompiling tables. These tables are used exclusively by the ADAPT I decompiler for performing language transformations from UDL to the four target host query languages. Appendix D provides the necessary information required for schema and tranfile maintenance.

## UDL TRANSFORMATIONS

The ADAPT UDL has been designed to present a uniform user interface to a variety of database management systems. The users of these database management systems represent many areas of interest and, therefore, represent a varied range of requests which must all be adequately satisfied through UDL and its associated software components. The database management systems (referred to here as target systems) provide their own query languages to enable the user to interrogate, display, and update the databases.

The initial objectives to be met by ADAPT and its respective language, UDL, can be divided into two categories.

In designing UDL as a pure language, it must be:

a. A single language.

b. Functionally complete and independent.

c. An evolutionary language with respect to its functional power; i. e., simple and complex versions of the same function should be representable by the same basic root command, the difference being only in the presence or absence of optional parameters.

In designing UDL as a language which can be translated into other languages, it must:

a. Be capable of accessing existing and potential target systems.

b. Present a uniform user interface.

c. Be constructed such that its data structures and statements can be mapped onto the target systems.

d. Provide a consistent interpretation of data structures and statements as they are applied across the various target systems.

The transformation of one language into many diverse languages is a difficult task, especially if this transformation sequence is to satisfy foregoing condition d. Ostensibly, it seems possible that many or most UDL functions could be satisfied by transformations alone. That is, the function itself is not processed locally in ADAPT, but is transformed to an appropriate target system statement sequence where the function is performed remotely. This notion is caused in most part by the assumption that all target system query languages are complete enough that each contains some hypothetical minimal function set that is sufficient to generate all UDL functions. Therefore, with a little ingenuity, a target system statement sequence can be found which can be mapped onto by any given UDL statement. Unfortunately, this is not the case. Great variability is present in the target system languages where many languages completely lack important UDL functions. This is particularly true with data manipulation and complex display functions. Therefore, the ADAPT design strategy can not rely solely on a language transformation technique and, in fact, this particular technique should be minimized as much as possible. If this is not done, it will be impossible to construct a UDL that is both powerful enough to satisfy the basic data base management requirements and one that can be applied consistently across a multitude of target systems. In order to meet this objective, language transformations are being performed in two areas only: interrogation and file updating. It is mandatory that these two functional areas be implemented through language transformations since ADAPT does not have direct access to the various network files. In addition, a limited transformation of UDL display functions is also required; i.e., those functions which cause the transmission of target system file information through the network for subsequent display in ADAPT.

In order to demonstrate that UDL can be interpreted consistently across various target system files, four transformation goals must be met:

a. Demonstrate that each major data structure of all target systems can be mapped onto a legal UDL data structure. Inconsistent interpretations of data structures based on a particular file or target system are not valid transformations.

b. Demonstrate that all data contained in these mapped files can be interrogated and updated via legal UDL statements. Using inconsistent interpretations of UDL statements to achieve this is not a valid transformation.

c. Demonstrate that the UDL statements which were used to satisfy the interrogation or updating of files for a given target system can be mapped onto functionally equivalent statements of that target system.

d. Demonstrate that all required UDL outputs are achievable through target system responses.

The material which follows supports the statement that UDL does indeed meet these four goals. Although ADAPT I, the initial phase of ADAPT, is only concerned with data base interrogation and limited data display, it seems appropriate here to demonstrate the full transformability of UDL with respect to both interrogation and update. Therefore, the examples which follow utilize a full UDL, a UDL which is much larger and more complete than the one which has been developed for ADAPT I. The DIAOLS/ISS transformations presented in this section assume a completely interactive host, even though the actual ADAPT I DIAOLS implementation assumes a batch host which DIAOLS will initially be on COINS II. Upon the existence of a local database manager in ADAPT as recommended for ADAPT IV, the transformability of UDL into other target systems becomes more attainable. Based on research conducted by Logicon, it was discovered that reliance upon a local database manager strengthened and expanded the overall transformability of UDL, even to somewhat unorthodox database query languages.

Four database management systems are discussed in the following: DIAOLS/ISS, DMS-1100/QLP, SOLIS, and TIPS/TILE. For each system, a simple hypothetical file is used to demonstrate the transformability of UDL to the various query languages.

## DIAOLS/ISS

OVERVIEW – The DIA On-Line System (DIAOLS) is an intelligence data handling system developed by the Defense Intelligence Agency (DIA). The primary reason for the development of DIAOLS in the late 1960s was to provide the intelligence analyst with direct access to the information he needs to do his job. As such, DIAOLS provides for user servicing in three modes: interactive time sharing, remote batch processing, and local batch processing.

DIAOLS is currently operational on two Honeywell G-635 computers supported by the General Comprehensive Operating Supervisor (GCOS), release F8. The two G-635s are called System One and System Two, and utilize primarily Honeywell 6000 series peripherals including DSU-191 disk storage. Three Datanet 30s currently provide the communications front-end processing for both systems. Terminals currently available include Model 37 ASR and KSR teletypewriters, Raytheon DIDS 400 CRT displays with Inktronic printers, and COPE 1200 remote batch terminals (RBTs).

A modified version of GCOS is used to support processing of classified information. System One supports compartmented processing to the Top Secret/SI/SAO level. System Two supports only non-compartmented processing. System One supports two software subsystems: the Intelligence Support System (ISS) and the Community On-Line Intelligence System (COINS). The ISS supports analyst requirements for information storage, retrieval, display, and maintenance in an interactive timesharing mode.

The COINS provides analysts with access to databases at other national agencies in a store and forward mode. System Two is basically a standard Honeywell system supporting several language subsystems including BASIC, FORTRAN, CARDIN, TEXT EDITOR, etc.

The ISS was designed as a generalized storage and retrieval system which would quickly allow analysts to seek out direct answers to specific questions. The key design factor was rapid response to unpredictable queries. Each element of a file designated as retrievable is equally likely to be used as a key for retrieval. The kinds of data entities supported by the ISS include elements, records, and files. The types of data allowed have an alphabetic, numeric, or alphanumeric format. The logical structuring of data in the ISS is hierarchical using inverted lists to provide rapid response. The storage structure used by the ISS is random within a file. A Random Access Management System (RAMS) performs all logical/physical input/output for ISS routines.

Data interrogation, processing, maintenance, updating, display, and report generation are provided online by the ISS. Data definition and database creation are provided via the local batch mode.

In order to illustrate the transformation of DIAOLS/ISS data structures to functionally equivalent UDL data structures, a simple file has been constructed (figure 1). Three records are shown in this file, each containing information concerning hypothetical targets. The following paragraphs examine this file showing the transformation of ISS data structures, and the mapping of UDL interrogation, display, and update statements to functionally equivalent DIAOLS/ISS statements.

## DIAOLS/ISS DATA DESIGN –

Data Structures – In figure 1, all ISS data structures are represented in UDL terminology. The ISS record maps directly onto a UDL record. ISS fixed and variable elements are represented as UDL single-valued

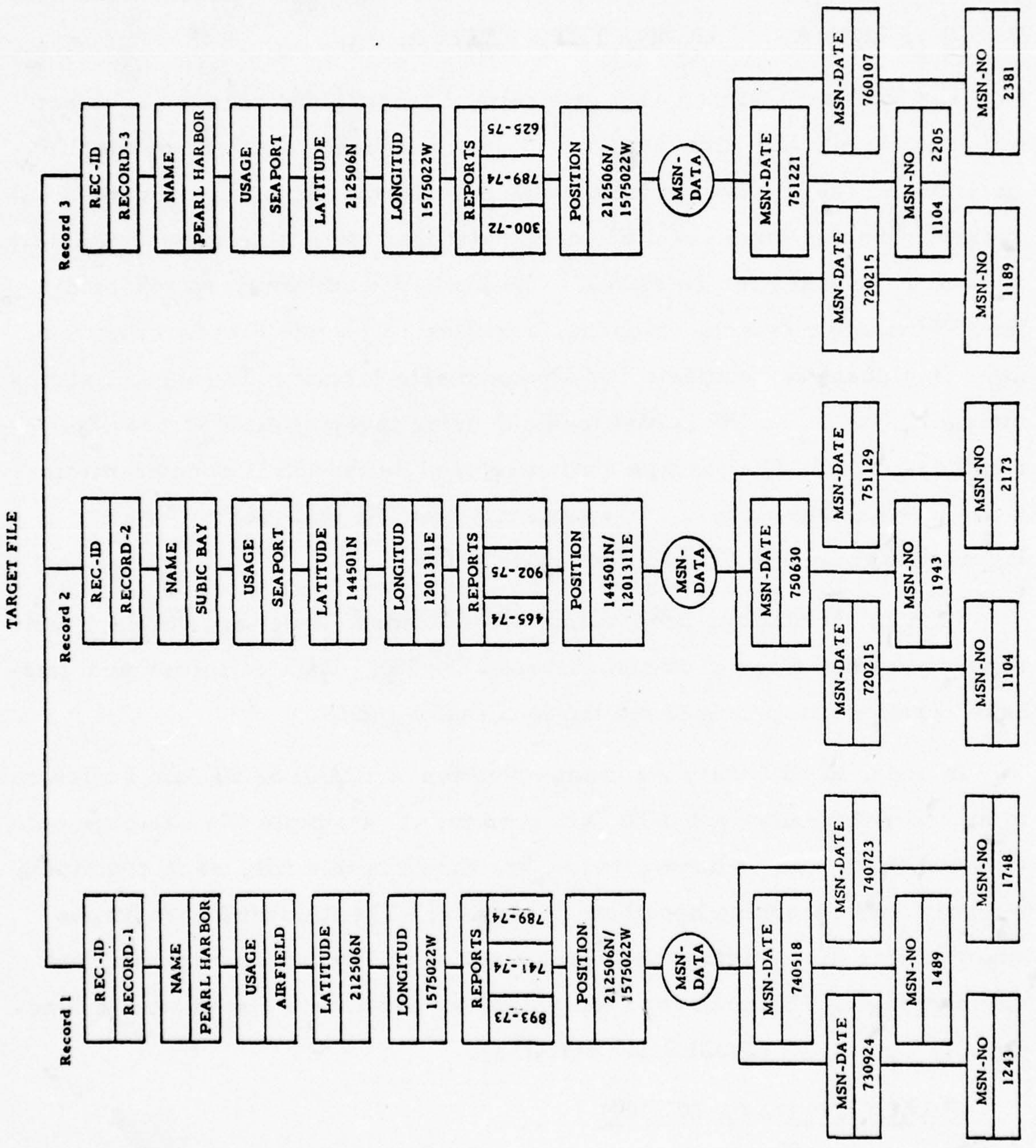


Figure 1. ISS Sample File

fields, such as the field USAGE. The ISS periodic element is represented in UDL as a multivalued field, such as REPORTS. Relational periodic elements, the most complex data structure in ISS, maps onto a UDL repeating group. Within the repeating group, the fields may be either single-valued or multivalued depending on the actual use of the relational periodic subscript in the ISS file. Single-valued and multivalued field examples are shown in figure 1. The internal subscripts used by ISS are equivalent to an occurrence of the repeating group. Figure 2 illustrates the repeating group MSN-DATA as it would exist in DIAOLS; it is functionally equivalent to its couterpart in figure 1. Notice that all value instances with the same index form an occurrence of the repeating group MSN-DATA in figure 1.

Data Types – The five data types described for UDL are present in DIAOLS. In figure 1, only three UDL data types are represented: geographic for field POSITION; numeric for fields MSN-NO and MSN-DATE; and alphanumeric for fields NAME, USAGE, LATITUDE, LONGITUD, MSN-DATE, and REPORTS. The choice of a data type for any given field of a target system file is somewhat subjective. For example, all DIAOLS group data structures will be mapped onto a UDL single-valued field even though in DIAOLS it is the juxtaposition of several simple elements. This field will then be assigned a data type designation of variable-length text. Lengthy variable elements in DIAOLS (nongroups) will be mapped onto a fixed-length text data type. Data type mappings for alphanumeric, numeric, and geographic are more straightforward, since data type designations are determined based on their usage in DIAOLS files.

Data Attributes – Of the seven data attributes recognized under UDL, five will be utilized for ISS files: keyed, dependent, visible, major, and nonmajor. This implies that all ISS fields are searchable (to some degree) and displayable. All inverted ISS elements are treated as keyed fields and noninverted elements as dependent fields. All ISS elements designated as

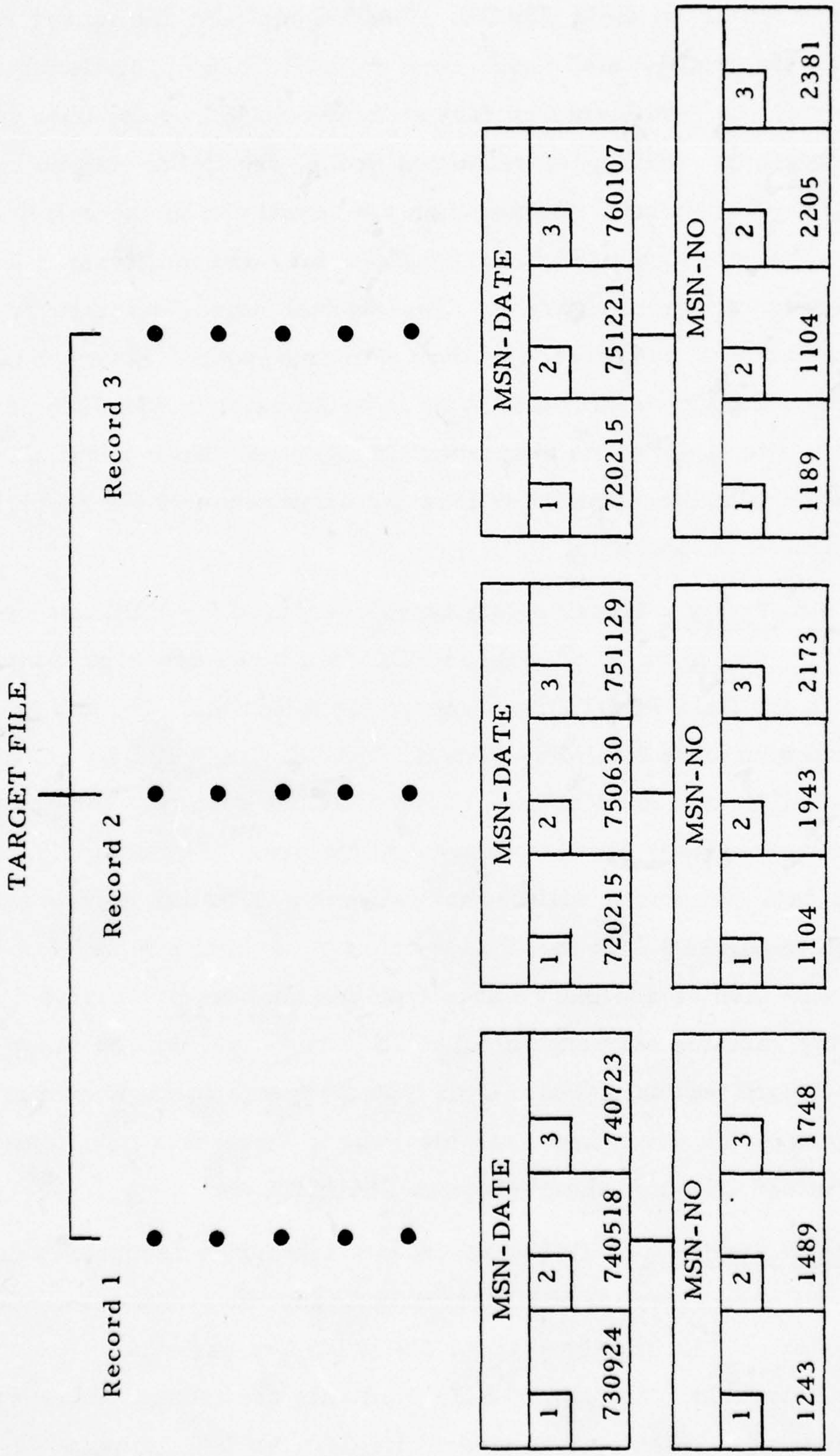


Figure 2. ISS Relational Periodic Element.

minimum data elements (MDEs) will have an update attribute of major. All non-MDEs will have a nonmajor update attribute.

INTERROGATION – This paragraph illustrates the transformation of UDL interrogation statements onto a set of functionally equivalent ISS statements. The sample database in figure 1 is used in conjunction with statement sequences in figures 3 through 15. Note, the transformations presented below assume an interactive DIAOLS host. In ADAPT I, DIAOLS is assumed to be batch.

Figure 3 shows a very simple example of a query which will retrieve all targets with a name of Pearl Harbor. The transformation from UDL to ISS instructions is very straightforward. Figure 4 takes this query one step further and specifies airfields at Pearl Harbor. Again, the transformation is straightforward. In figure 5, the first FIND statement, identified by "labell," isolates those targets which are a seaport. The second

```
UDL Statements

      labell FIND IN TARGET NAME EQ 'PEARL HARBOR';
      [NUMBER OF RECORDS FOUND = 2]

ISS Statements
      REQUEST NEW
      [ENTER QUERY]
      NAME = :PEARL HARBOR:.
      [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
                  TARGET                          2
       ACTION
```

Figure 3. UDL/ISS Transformation (Interrogation).

UDL Statements

```
label1 FIND IN TARGET NAME EQ 'PEARL HARBOR';

[NUMBER OF RECORDS FOUND = 2]

label2 FIND SOURCE (label1) USAGE EQ 'AIRFIELD';

[NUMBER OF RECORDS FOUND = 1]
```

ISS Statements

```
REQUEST NEW

[ENTER QUERY]

NAME = :PEARL HARBOR:.

[NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
[          TARGET                            2]
[ACTION                                       ]

REFINE NEW

[ENTER QUERY]

USAGE = :AIRFIELD:.

[NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
[          TARGET                            1]
[ACTION                                       ]
```

Figure 4. UDL/ISS Transformation (Interrogation).

UDL Statements

```
label1 FIND IN TARGET USAGE EQ 'SEAPORT';
[NUMBER OF RECORDS FOUND =-2]
label2 FIND SOURCE (label1) MSN-DATA (MSN-DATE EQ 720215
AND MSN-NO EQ 1104);
[NUMBER OF RECORDS FOUND = 1]
```

ISS Statements

```
REQ NEW
[ENTER QUERY]
USAGE = :SEAPORT:.
[NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST
          TARGET                          2
ACTION                                       ]
RELATE NEW
[ENTER QUERY]
MSN-DATE = :720215: TO MSN-NO = :1104:.
[NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST
          TARGET                          1
ACTION                                       ]
```

Figure 5. UDL/ISS Transformation (Interrogation).

FIND statement, a dependent FIND identified by "label2," refines the first FIND statement by applying additional selection criteria against those records already isolated. This statement is more complicated than the first one. Here, all targets with MSN-DATE 720215 and MSN-NO 1104 are isolated. It should be noted that the scope-qualifier of MSN-DATA forces the selection to guarantee that the mission date and mission number specifications be satisfied for the same occurrence of the repeating group MSN-DATA. This statement reduces the selected record set to a single record, record 2. If the scope-qualifier had not been used, both record 2 and record 3 would have satisfied the selection criteria since they both have MSN-DATEs of 720215 and MSN-NOs of 1104. However, only record 2 has MSN-NO 1104 performed on MSN-DATE 720215. Again, this example provides a straightforward mapping onto commands acceptable to the ISS. The RELATE statement in the ISS is equivalent to the scope-qualifier in UDL and is utilized in a dependent mode to operate over relational periodic elements.

Search conditions against keyed fields may be combined as in figure 6. This query is functionally equivalent to the one in figure 4 and will isolate the same record, record 1.

UDL Statements

```
    label1 FIND IN TARGET NAME EQ 'PEARL HARBOR' AND
    USAGE EQ 'AIRFIELD';
    [NUMBER OF RECORDS FOUND = 1]
```

ISS Statements

```
    REQ NEW
    [ENTER QUERY]
    NAME = :PEARL HARBOR: AND USAGE = :AIRFIELD:.
    [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST
                 TARGET                    1
     ACTION                                        ]
```

Figure 6. UDL/ISS Transformation (Interrogation).

Figure 7 is an example of a keyed query against the LATITUDE and LONGITUD fields. Once again the mapping is very straightforward.

Figure 8, however, poses a more complex problem. The second dependent FIND statement specifies that the circle search be performed. This may be mapped onto ISS statements in two ways. Either the ISS action GEO or CIRCLE may be requested and both will retrieve the same record (CIRCLE is used here). The only functional difference in terms of ISS is that the action CIRCLE calculates the radius distance for records lying within the circle specified. The point is that in UDL terms it must be defined ahead of time which ISS action to map to, CIRCLE or GEO. Dependent interrogations for the ISS, as shown in figures 4, 5, 7, and 8, act as "AND operations." If the two main constituents of a UDL FIND statement are connected with an "OR condition," the corresponding mapping to ISS is quite different and becomes very complicated. In reality, this will be a valid user

<u>UDL Statements</u>

```
    label1 FIND IN TARGET LATITUDE EQ '212506N' AND
    LONGITUD EQ '1575022W';
    [NUMBER OF RECORDS FOUND = 2]
```

<u>ISS Statements</u>

```
    REQ NEW
    [ENTER QUERY]
    LATITUDE = :212506N: AND LONGITUDE = :1575022W:.
    [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
    |        TARGET                           2   |
    [ACTION                                       ]
```

Figure 7. UDL/ISS Transformation (Interrogation).

<u>UDL Statements</u>

```
    label1 FIND IN TARGET USAGE EQ 'SEAPORT';
    [NUMBER OF RECORDS FOUND = 2]
    label2 FIND SOURCE (label1) POSITION INSIDE CIRCLE (100,
    %21 N/157 W);
    [NUMBER OF RECORDS FOUND = 1]
```

<u>ISS Statements</u>

```
    REQ NEW
    [ ENTER QUERY]
    USAGE = :SEAPORT:.

    ┌NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST┐
    │          TARGET                        2   │
    └ACTION                                      ┘

    CIRCLE NEW
    [ENTER COORDINATE NAMES]
    LATITUDE, LONGITUDE
    [ENTER QUERY]
    IF INSIDE CIRCLE-1.
    [FOR CIRCLE-1 ENTER LATITUDE, LONGITUDE, AND RADIUS]
    21N, 157W, 100
    [OPTION ? L = LIST, M = MODIFY, N = RESEQUENCE,
    R = RUN, S = SAVE]
    R

    ┌NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST┐
    │          TARGET                        1   │
    └ACTION                                      ┘
```

Figure 8. UDL/ISS Transformation (Interrogation).

requirement in only rare cases, and this description does not go into the procedure at this time.

ISS DISPLAY – Transformations from UDL to target systems for display operations require only that the desired fields be available from the target system. Data formatting and the various UDL print control operations are not mapped onto target system statements but are performed entirely in ADAPT. The ISS action OUTPUT will be mapped onto from UDL to output the desired fields which will then be manipulated in ADAPT.

Figure 9 gives an example of how to obtain the entire record which will be displayed with field name labels. This is the simplest means of displaying data. An example of the UDL formatting capability is shown in figure 10. For each record selected (record 1 in this case), the target name and position will be printed under the column titles "TARGET NAME" and "POSITION," printed left-justified at column 10 and right-justified to column 60, respectively. Two lines are skipped and the target name and position are printed left-justified and right-justified to columns 10 and 60, respectively.

ISS UPDATE – This paragraph discusses the transformation of UDL update statements to equivalent ISS statement sequences. One requirement of ISS update sequences is that the record-id of the record to be modified must be specified. The record-id is unique for each record, but it is usually functional data and not just some machine reference number (which the example in figure 1 unfortunately is). In most cases, the user will know the record-id of the record to be modified. Where the user does not know the record-id or where many records are to have the same modification, a keyed field may be used as the source and identification of records to be modified.

A new target record is added to the file in figure 11. Note that the UDL field POSITION is broken down into two ISS elements, LATITUDE and LONGITUDE. Also note that, since the mission date 730127 is the first

UDL Statements

```
label1 FIND IN TARGET NAME EQ 'PEARL HARBOR' AND
USAGE EQ 'AIRFIELD';
[NUMBER OF RECORDS FOUND = 1]
DISPLAY SOURCE (label1);
    [... RECORD HEADER ...
     REC-1D = RECORD-1
     NAME = PEARL HARBOR
     USAGE = AIRFIELD
     :
     MSN-DATA
         MSN-DATE = 730924
         MSN-NO = 1243
     :                      ]
```

ISS Statements

```
REQ NEW
[ENTER QUERY]
NAME = :PEARL HARBOR: AND USAGE = :AIRFIELD:.
[NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST
          TARGET                   1
 ACTION                                       ]
OUTPUT S; ALL.
    [RECORD-1D = RECORD-1
     NAME = PEARL HARBOR
     :                      ]
```

Figure 9. UDL/ISS Transformation (Display).

UDL Statements

```
labell FIND IN TARGET NAME EQ 'PEARL HARBOR' AND
USAGE EQ 'AIRFIELD';
[NUMBER OF RECORDS FOUND = 1]
DISPLAY SOURCE (labell) 'TARGET NAME', 'POSITION',
NAME, POSITION;
FORMAT (PAGE, AT (10), TO (60), SKIP (2), AT (10),
TO (60));

  10                                        60
  TARGET NAME                         POSITION
  PEARL HARBOR                 212506N/1575022W
```

ISS Statements

```
REQ NEW
[ENTER QUERY]
NAME = :PEARL HARBOR: AND USAGE = :AIRFIELD:.

 NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST
          TARGET                        1
 ACTION

OUTPUT S; NAME, LATITUDE, LONGITUDE.
```

Figure 10. UDL/ISS Transformation (Display).

UDL Statements

```
        CREATE IN TARGET RECORD-ID EQ 'RECORD-4',
        NAME EQ 'SASEBO',
        USAGE EQ 'SEAPORT', POSITION EQ %33 10 31 N/
        129 43 38 E,
        MSN-DATA (MSN-DATE EQ 730127, MSN-NO EQ 1199);
```

ISS Statements

```
        ADD
        [LISTS MDE + INSTRUCTIONS]
        :RECORD-4:, NAME :SASEBO:,
        USAGE :SEAPORT:, LATITUDE :331031N:, LONGITUDE
        :1294338E:, MSN-DATE ADD 01:730127:, MSN-NO ADD 01:1199:.
        [RECORD ADDED]
```

Figure 11. UDL/ISS Transformation (Update).

(and only) occurrence for the repeating group MSN-DATA, a subscript of 01 is assigned in the ISS ADD statement for all applicable entries.

Record 1 is removed from the file in figure 12. In the case of a DELETE, the record-ids must be specified. Therefore, the ISS sequence must output the record-id so that ADAPT can use it to actually delete the record.

As shown in figure 13, changes to existing records which do not reference repeating group fields are quite simple. UDL statements map easily onto the ISS change action.

If repeating groups are referenced as in figure 14 or complicated and dependent FIND statements are used as in figure 15, the transformation becomes much more complicated.

```
UDL Statements

     label1 FIND IN TARGET REC-ID EQ 'RECORD-1';
     [NUMBER OF RECORDS FOUND = 1]
     REMOVE SOURCE (label1);

ISS Statements

     REQ NEW
     [ENTER QUERY]
     RECORD-ID = :RECORD-1:.
     ┌NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST┐
     │        TARGET                        1      │
     └ACTION                                       ┘
     OUTPUT S;.
     [RECORD-ID = RECORD-1]
     DELETE
     [INSTRUCTIONS]
     :RECORD-1:.
```

Figure 12. UDL/ISS Transformation (Update).

In figure 14, the records selected must be output so that the actual subscripts of the ISS relational periodics referenced by the UDL repeating groups can be determined. Without the subscripts, a mapping onto ISS is not possible. In the figure, the appropriate subscript is 03.

In figure 15, the records must be output to determine the record-ids to be used in the ISS CHANGE sequence, and to determine the first available subscript for the relational periodic entries to be made from the UDL repeating group. 04 was the first subscript available in this case.

UDL Statements

```
labell FIND IN TARGET NAME EQ 'SUBIC BAY';
[NUMBER OF RECORDS FOUND = 1]
CHANGE SOURCE (labell) LATITUDE to '144500N', REPORTS
EQ '465-74' TO '464-74';
```

ISS Statements

```
REQ NEW
[ENTER QUERY]
NAME = : SUBIC BAY:.
[NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
[          TARGET                         1  ]
[ACTION                                      ]
OUT S; RECORD-ID.
[RECORD-ID = RECORD-2]
CHANGE
[INSTRUCTIONS]
:RECORD-2:, LATITUDE:144500N:, REPORTS CHANGE:465-74:
TO :464-74:
[CHANGES ACCEPTED]
```

Figure 13. UDL/ISS Transformation (Update).

```
UDL Statements

    label1 FIND IN TARGET REC-ID EQ 'RECORD-3';

    [NUMBER OF RECORDS FOUND = 1]

    CHANGE SOURCE (label1) OCC (MSN-DATE EQ 760107) MSN-NO

    EQ 2381 TO 2380;

ISS Statements

    REQ NEW

    [ENTER QUERY]

    RECORD-ID = :RECORD-3:.

    [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
    [         TARGET                         1    ]
    [ACTION                                       ]

    OUT S; MSN-DATE, MSN-NO, RECORD-ID.

    [OUTPUT LIST                                  ]
    [ACTION                                       ]

    CHANGE

    [INSTRUCTIONS]

    :RECORD-3:, MSN-NO CHANGE 03:2381: TO 03:2380:.

    [CHANGES ACCEPTED]
```

Figure 14. UDL/ISS Transformation (Update).

```
UDL Statements
    label1 FIND IN TARGET USAGE EQ 'SEAPORT';
    [NUMBER OF RECORDS FOUND = 2]
    label2 FIND SOURCE (label1) MSN-DATA (MSN-DATE EQ 720215
    AND MSN-NO EQ 1104);
    [NUMBER OF RECORDS FOUND = 1]
    ADD SOURCE (label2) MSN-DATA (MSN-DATE EQ 760325,
    MSN-NO EQ 3105);

ISS Statements
    REQ NEW
    [ENTER QUERY]
    USAGE = :SEAPORT:.
    [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
    [           TARGET                       2   ]
    [ACTION                                      ]
    REL NEW
    [ENTER QUERY]
    MSN-DATE = :720215: TO MSN-NO = :1104:
    [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
    [          TARGET                        1   ]
    [ACTION                                      ]
    OUTPUT S; MSN-DATE, MSN-NO, RECORD-ID.
    [OUTPUT LIST   ]
    [      :       ]
    [ACTION        ]
    CHANGE
    [INSTRUCTIONS]
    :RECORD-2:, MSN-DATE ADD 04:760325:, MSN-NO
    ADD 04:3105:.
    [CHANGES ACCEPTED]
```

Figure 15. UDL/ISS Transformation (Update).

## DMS-1100/QLP

OVERVIEW – Data Base Management System 1100 (DMS-1100) is UNIVAC's version of CODASYL's DBTG Data Base Management System. The basic premise of the CODASYL Data Base Task Group is that there must be a separation of data description functions from data access functions. This implies a distinction between the database manager, whose function is to describe the data records, and the user, whose function is to access the data. This distinction leads to the separate development of a data description language and a data manipulation language.

Currently, DMS-1100 is operational with both the Data Definition Language (DDL) and the Data Manipulation Language (DML). In addition, a new interactive Query Language Processor (QLP) has been recently developed by Univac. The DMS-1100 system resides on the Univac 1100 hardware series where it operates under the EXEC-8 operating system.

DMS-1100 provides great flexibility to the user where simple sequential organization of data to complicated network structures is allowed. These data structures are divided into areas (files), records, and items (fields). In addition, a user may optionally define set relationships over records, where each set is called by name. The record location-mode can be specified by the user where direct, index sequential, calculation by procedure (CALC) and via-set are allowed. A DDL is used by the user to define his databases.

DMS-1100 provides the user with a DML which is embedded in a Univac 1100 COBOL environment. This language allows the user to interrogate his database (FIND, FETCH, and GET commands), to update or restructure his database (STORE, MODIFY, DELETE, INSERT, and REMOVE commands), and to conditionally control the sequencing of his DML procedure (IF command). In addition, other miscellaneous data manipulation, database control, and run-unit control commands are provided, such as OPEN, CLOSE, MOVE, IMPART, and DEPART. The display capability is provided by the standard COBOL display commands. The online interactive QLP provides essentially the same capabilities as those previously described for the DML. It is this interactive query language that UDL is transformed to.

To illustrate the transformation of DMS-1100 data structures to functionally equivalent UDL data structures, a simple file has been constructed (figure 16). Three records are shown in this file, each containing information concerning a hypothetical employee file. Each record contains fields depicting an employee's name, job position, salary, spouse's name, birthdate, and information concerning the employee's children. The following paragraphs examine this file to show the transformation of DMS-1100 data structures to equivalent UDL data structures, and the mapping of UDL interrogation, display, and update statements to functionally equivalent QLP statements.

## DMS-1100 DATA DESIGN –

Data Structures – In figure 16, all major DMS-1100 record data structures are represented in UDL terminology. DMS-1100 areas map onto UDL files. Similarly, the DMS-1100 record maps directly onto a UDL record. DMS-1100 data-items are represented as UDL single-valued fields; i.e., NAME, TITLE, SALARY, and SPOUSE. DMS-1100 array structures are shown with UDL arrays CNAME and PKIND. For the array CNAME, the field CHILDNAM and the array PKIND are defined, and for the array PKIND the field PETS is defined. CHILDNAM and PETS are single-valued fields.

The DMS-1100 UDL data structure mapping is quite straightforward.

Data Types – Of the five data types recognized in UDL, only two are utilized for DMS-1100 files: alphanumeric and numeric. Both of these data types are illustrated in figure 16, where fields NAME, TITLE,

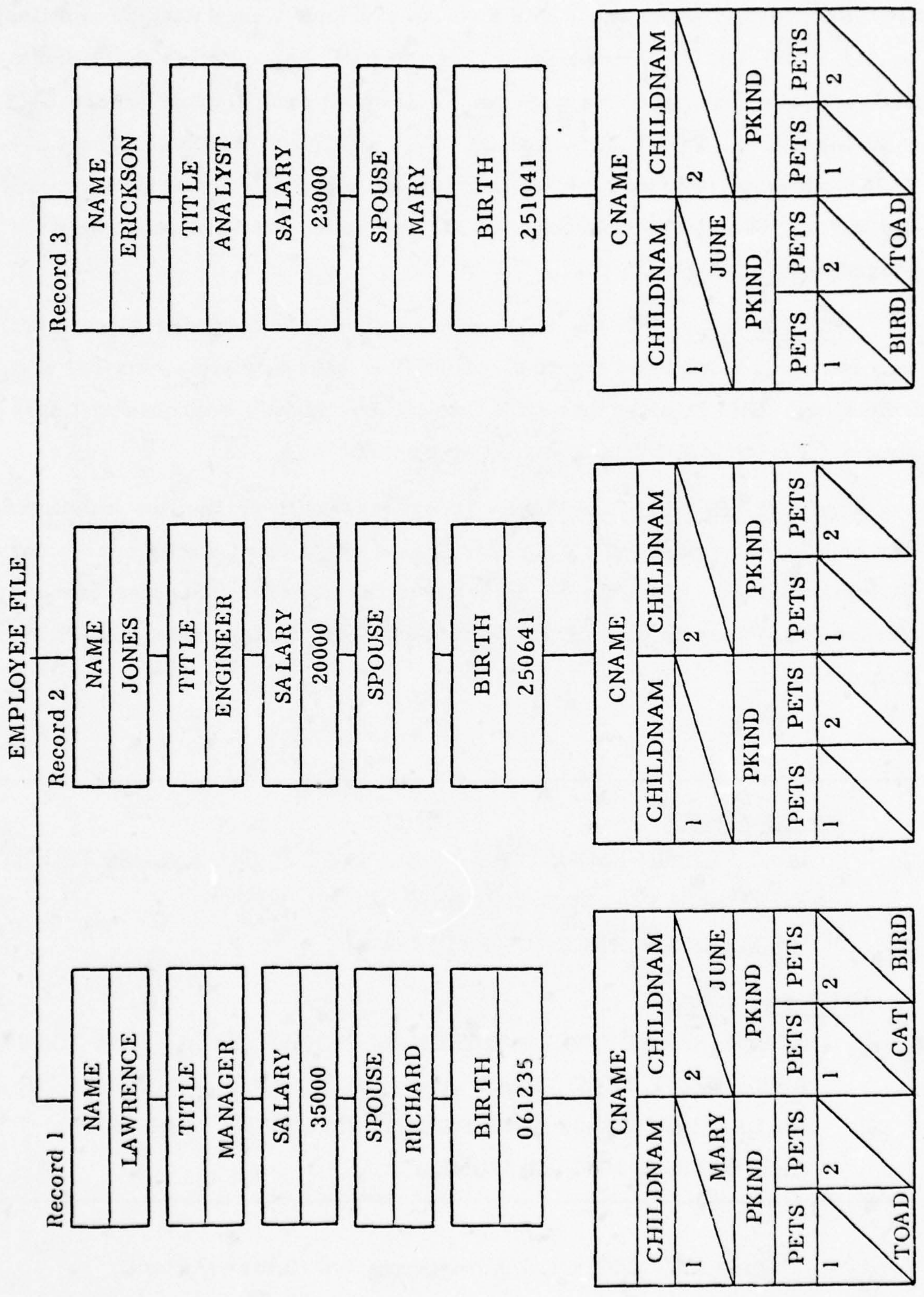


Figure 16. DMS-1100 Sample File.

SPOUSE, CHILDNAM, and PETS have an alphanumeric data type and fields SALARY and BIRTH have a numeric data type. This implies that certain UDL operators can not be applied in FIND statements that reference DMS-1100 files; i.e., CONTAINS, INSIDE, OUTSIDE, and ALONG. The CONTAINS operator is valid only for variable or fixed length text data types, and the INSIDE, OUTSIDE, and ALONG operators are applicable only with geographic data types.

Data Attributes – Of the seven data attributes recognized under UDL, only four will be utilized for DMS-1100 files: keyed, visible, major, and nonmajor. This implies that all fields can be directly searched and displayed. Field NAME is designated as major.

DMS-1100 INTERROGATION – This paragraph describes the transformation of UDL interrogation statements onto a set of functionally equivalent QLP statements. The sample database utilized in the foregoing paragraph *(figure 16) is used in conjunction with the statement sequences shown in* figure 17.

UDL Statements

```
    label FIND IN EMPLOYEE (SALARY GT 20000) AND (PETS(1, 1)
    EQ 'TOAD' XOR NOT CHILDNAM(2) EQ 'JUNE');
    [NUMBER OF RECORDS FOUND = 2]
```

QLP Statements

```
    COUNT SALARY PETS CHILDNAM WHERE (SALARY GT '20000'
    AND PETS(1, 1) EQ 'TOAD' AND CHILDNAM(2) EQ 'JUNE') OR
    (SALARY GT '20000' AND PETS(1, 1) NOT EQ 'TOAD' AND
    CHILDNAM(2) NOT EQ 'JUNE')
```

Figure 17. UDL/QLP Transformation (Interrogation).

Consider the UDL statement sequence shown in figure 17. The UDL FIND statement isolates those employees whose salary exceeds $20,000, where this employee either does not have a second child named June or does have a first child that owns a pet toad, but not both conditions. Employees Lawrence and Erickson satisfy this FIND statement (records 1 and 3).

The functionally equivalent QLP statements are shown in figure 17. The QLP COUNT statement is used to get the number of records satisfying the selection criteria specified in the where-clause. Since QLP does not have an exclusive-or operator (XOR), a logically equivalent expression using AND, OR, and NOT operators is used. The QLP COUNT statement provides counts of records for each field named in the specified field list.

The UDL/QLP interrogation statement transformation is quite straightforward.

QLP DISPLAY – This paragraph describes the transformations required for requesting partial and whole records from DMS-1100 for subsequent display through UDL. Figure 16 is used as the sample database for illustrating these transformations.

Consider the UDL statement sequence in figure 18. The FIND statement isolates two employee records, record 1 and record 3. The DISPLAY statement outputs the employee's name and the employee's spouse's name for each of these records in the UDL default format. Although in UDL a user can reference a previous interrogation statement remotely via a statement label, the selection criteria must be respecified in the where-clause for the QLP LIST statement. For the default case where the display-list is not specified in the UDL DISPLAY statement, UDL will output the entire record. In this case, all fields must be specified in the QLP LIST statement.

QLP UPDATE – This paragraph discusses the transformation of UDL update statements to equivalent QLP statement sequences. As in the

UDL Statements

```
label FIND IN EMPLOYEE (SALARY GT 20000) AND (PETS(1, 1)
EQ 'TOAD' XOR NOT CHILDNAM(2) EQ 'JUNE');
[NUMBER OF RECORDS FOUND =2]
DISPLAY SOURCE (label) NAME, SPOUSE;

    ... record header ...
    NAME = LAWRENCE
    SPOUSE = RICHARD
    ... record header ...
    NAME = ERICKSON
    SPOUSE = MARY
```

QLP Statements

```
COUNT SALARY PETS CHILDNAM WHERE (SALARY GT '20000'
AND PETS(1, 1) EQ 'TOAD' AND CHILDNAM(2) EQ 'JUNE')
OR (SALARY EQ '20000' AND PETS(1, 1) NOT EQ 'TOAD'
AND CHILDNAM(2) NOT EQ 'JUNE')

LIST NAME SPOUSE WHERE (SALARY GT '20000' AND PETS(1,1)
EQ 'TOAD' AND CHILDNAM(2) EQ 'JUNE') OR (SALARY GT
'20000' AND PETS(1, 1) NOT EQ 'TOAD' AND CHILDNAM(2)
NOT EQ 'JUNE')
```

Figure 18. UDL/QLP Transformation (Display).

preceding, the sample data base presented in figure 16 is used in conjunction with the statement sequences shown in figures 19 through 21.

Of the five UDL update statements available, only three can be used on DMS-1100 files. Since DMS-1100 does not provide true multiply occurring data structures below the record level, the UDL ADD or DELETE statements are not legal. Although DMS-1100 provides array structures which may contain multiple values, array structure sizes are fixed at definition time, hence they can not be expanded or reduced.

UDL Statements

```
    label FIND IN EMPLOYEE (SALARY GT 20000) AND (PETS(1, 1)
    EQ 'TOAD' XOR NOT CHILDNAM(2) EQ 'JUNE');
    [NUMBER OF RECORDS FOUND = 2]
    REMOVE SOURCE (label);
```

QLP Statements

```
    COUNT SALARY PETS CHILDNAM WHERE (SALARY GT '20000'
    AND PETS(1, 1) EQ 'TOAD' AND CHILDNAM(2) EQ 'JUNE')
    OR (SALARY GT '20000' AND PETS(1, 1) NOT EQ 'TOAD' AND
    CHILDNAM(2) NOT EQ 'JUNE')

    DELETE EMP WHERE (SALARY GT '20000' AND PETS(1, 1) EQ
    'TOAD' AND CHILDNAM(2) EQ 'JUNE') OR (SALARY GT
    '20000' AND PETS(1, 1) NOT EQ 'TOAD' AND CHILDNAM(2)
    NOT EQ 'JUNE')
```

Figure 19. UDL/QLP Transformation (Update).

UDL Statements

```
CREATE IN EMPLOYEE NAME EQ 'JOHNSON', TITLE EQ
'PROGRAMMER', SPOUSE EQ 'MARY', BIRTH EQ 070645,
CHILDNAM(1) EQ 'WILLIAM', PETS(1, 1) EQ 'DOG';
```

QLP Statements

```
CREATE EMP WITH NAME EQ 'JOHNSON', TITLE EQ
'PROGRAMMER', SPOUSE EQ 'MARY', BIRTH EQ '070645',
CHILDNAM(1) EQ 'WILLIAM', PETS(1, 1) EQ 'DOG'
```

Figure 20. UDL/QLP Transformation (Update).

UDL Statements

```
label FIND IN EMPLOYEE NAME EQ 'ERICKSON';
[NUMBER OF RECORDS FOUND = 1]
CHANGE SOURCE (label) SALARY TO 26000, CHILDNAM(2)
TO 'MICHAEL', PETS(2, 1) TO 'HORSE';
```

QLP Statements

```
COUNT NAME WHERE NAME EQ 'ERICKSON'
CHANGE SALARY EQ '26000' CHILDNAM(2) EQ 'MICHAEL'
PETS(2, 1) EQ 'HORSE' WHERE NAME EQ 'ERICKSON'
```

Figure 21. UDL/QLP Transformation (Update).

In figure 19, all employees who satisfy the conditions specified in the FIND statement are removed from the employee file. The corresponding QLP statement sequence must repeat the selection criteria in the DELETE statement since QLP does not provide a facility for remotely specifying interrogation sequences. In figure 20, a new employee named JOHNSON is added to the employee file. The transformation to an equivalent QLP statement sequence is very straightforward, where the same comments concerning figure 19 also apply.

In figure 21, employee Erickson's salary is changed from $23,000 to $26,000 and a new child and its associated pet is added. Although a new child and pet are added, the array position (index 2) was already available, hence the modification can be accomplished with the UDL CHANGE statement. In figure 16, the arrays CNAME and PKIND have two index positions each; therefore, a user could not add a third child or pet.

## SOLIS

<u>OVERVIEW</u> – Even though the SIGINT On-Line Information System (SOLIS) is a document retrieval system, the data structures of SOLIS follow the standard structures defined in UDL. Therefore, the interrogation of the SOLIS data base is easily accomplished in UDL.

A sample document file has been described for SOLIS in UDL format and terminology and is shown in figure 22. There are three records (documents) in the file, each with a title, message number, date, and text. An additional field is shown called TEXTTERM which contains the actual key terms in the text of the document which can be used in forming queries.

An important point to notice in this sample database is the mapping of the field names used in UDL to the actual retrieval strategies in SOLIS. For example, the field called TITLE in UDL actually corresponds to

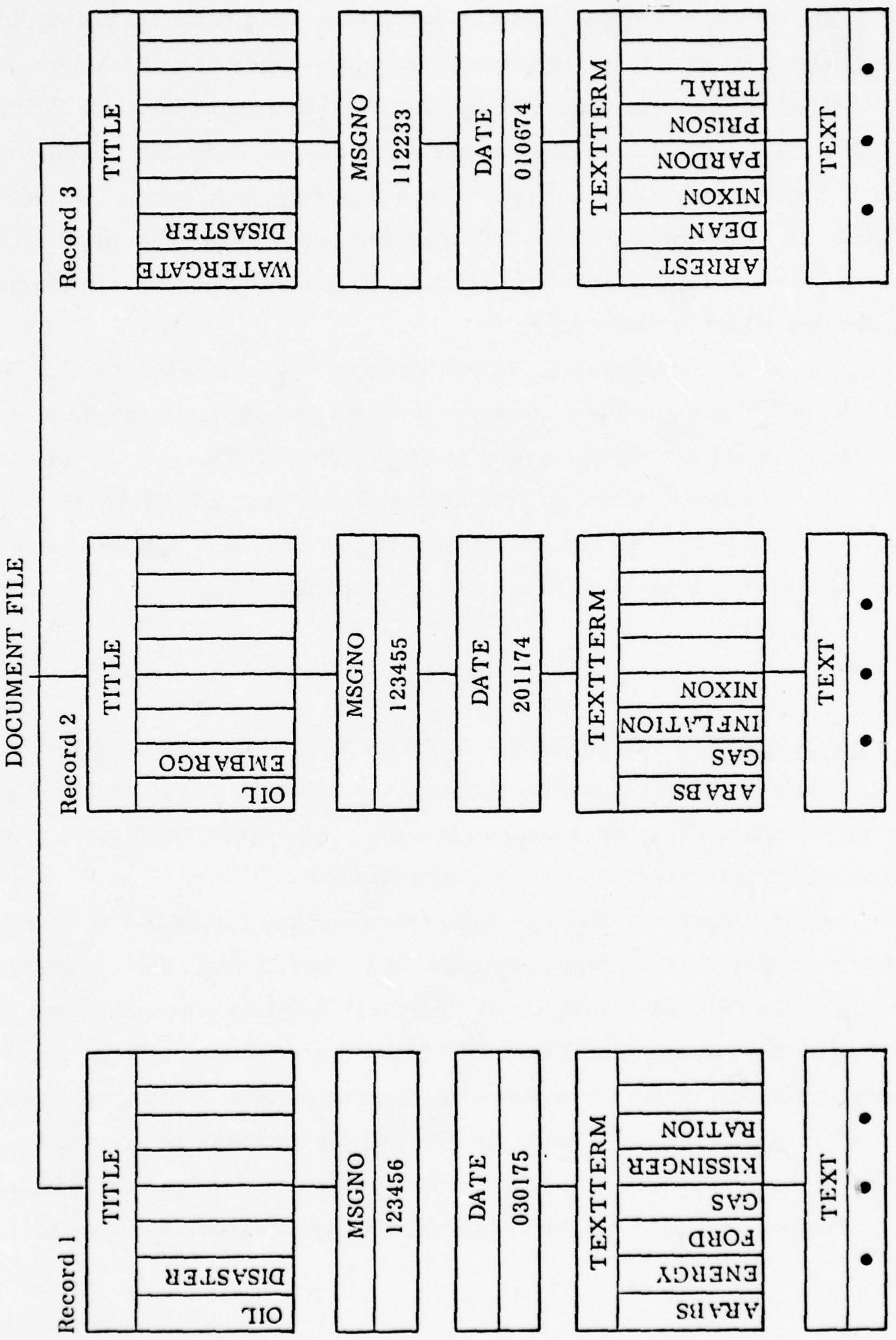


Figure 22. SOLIS Sample File

TILDEX in SOLIS. This mapping provides the uninitiated user with a more meaningful retrieval strategy than provided in SOLIS.

SOLIS provides online editing, storage, and retrieval of NSA end-product documents. The most current six months of documents are online with the newest document being at least 15 minutes old. Documents are retrieved using terms, recognizable by SOLIS as index terms, which are kept in one or more system dictionaries. SOLIS has been operational since November, 1972.

SOLIS resides on the Burroughs B-6700 computer system. The remote terminals are Burroughs B-9353 CRTs and teletypewriters for low-speed, hardcopy output. High-speed hardcopy output takes place near the B-6700 on high-speed printers.

Messages enter the SOLIS system in a raw, unretrievable form. Each message is filed, edited, and made retrievable. Old messages are automatically phased out of SOLIS as new messages come in. All editing and retrieval functions are handled by a software package called the SOLIS/MESSENGER. SOLIS physical data structures are organized by volume, file, record, and word.

The logical data entities are terms from the document and the documents themselves.

Document retrieval and display are accomplished by the user entering the parts of his request on a formatted CRT screen. Basically, key words and/or other document descriptors (such as date of entry into SOLIS) are entered on the CRT and the Send key is depressed. The response from SOLIS is the number of documents satisfying the request. To then view the text of the documents retrieved, the user need only press the Send key again and the zero page of the message will appear.

A guide to the sample field name-retrieval strategy mapping is shown in figure 23.

| UDL Field Names | SOLIS Retrieval Strategy | Meaning |
|---|---|---|
| TITLE | TILDEX | (Long title) |
| MSGNO | S | (Serial number) |
| DATE | D | (Date) |
| TEXTTERM | WINDEX | (Terms from the message text) |

Figure 23. SOLIS Retrieval Strategy Mappings.

Detailed descriptions of the UDL data structures found in SOLIS and queries to this database follow.

SOLIS DATA DESIGN –

Data Structures – There are two UDL field structures found in SOLIS, both illustrated in the sample database (figure 22). One is the single-valued field and is found in the fields named MSGNO, DATE, and TEXT. The other is the multivalued field represented by the fields TITLE and TEXTTERM.

The record data structure in UDL corresponds to the document structure in SOLIS. In other words, when the user, using UDL, selects a record from the SOLIS database, he is actually selecting a SOLIS document. UDL will also map individual files onto the various volumes that are distinguished in SOLIS. UDL files correspond to SOLIS file/volume pairs.

Data Types – All the fields in the SOLIS sample file, except DATE, are considered to be character type; specifically, alphanumeric. DATE is a numeric field.

Data Attributes – In SOLIS, only the actual document text and a few preamble fields can be displayed and, therefore, the only visible fields are TEXT and MSGNO. But it is possible to query the system based on selected parts of each document, such as the title (TITLE), the message number (MSGNO), the date (DATE), or terms in the text of the document (TEXTTERM). Therefore, these fields are keyed. The TEXT field is nonkeyed because not all terms in the text of the document can be used in selection criteria; i. e., only those in the field TEXTTERM can be used in selection criteria. Figure 24 shows the fields from the sample file and their attributes.

SOLIS INTERROGATION – Figure 25 illustrates how a UDL FIND statement would access the SOLIS document file (figure 22) and how this would be accomplished via translation to the SOLIS query language format. The problem is to determine how many records (documents) satisfy the condition that the term GAS is in the text of the document but the term DISASTER does not appear in the title of the document. One record is selected (record 2) by this FIND statement, and the UDL response indicates this (note the response in brackets). In all cases of mapping from UDL to SOLIS, the free screen will be established prior to transmitting the SOLIS query. The response to the SOLIS query is the number of documents satisfying the query (i. e., "1 SIGINT ITEMS SATISFY THIS QUERY"), and this obviously can be used directly to generate the appropriate UDL response.

| Field | Attributes |
|---|---|
| TITLE | Keyed, invisible |
| MSGNO | Keyed, visible |
| DATE | Keyed, visible |
| TEXTTERM | Keyed, invisible |
| TEXT | Nonkeyed, visible |

Figure 24. SOLIS Field Attributes.

```
UDL Statements

      FIND IN DOCUMENT TEXTTERM EQ 'GAS' AND NOT TITLE
      EQ 'DISASTER';
      [NUMBER OF RECORDS FOUND = 1]

SOLIS Statements

      (free screen established)
      ; GAS ; - ALL WITHOUT (TILDEX DISASTER)
```

Figure 25. UDL/SOLIS Transformation (Interrogation).

SOLIS DISPLAY – Figure 26 includes the same query as in the interrogation example. The task is to display all the records selected (namely one), and of these records to display the field named TEXT. The output from this UDL DISPLAY statement is shown in the large brackets, and is

```
UDL Statements

      label FIND IN DOCUMENT TEXTTERM EQ 'GAS' AND NOT
      TITLE EQ 'DISASTER';
      [NUMBER OF RECORDS FOUND = 1]
      DISPLAY SOURCE (label) TEXT;
      [... record header ...          ]
      [TEXT = ... text of document    ]

SOLIS Statements

      (free screen established)
      ; GAS ; - ALL WITHOUT (TILDEX DISASTER)
      (ETX character used to retrieve each page of document found)
```

Figure 26. UDL/SOLIS Transformation (Display).

obtained from SOLIS by transmitting an ETX character. This character will retrieve the entirety of each document page by page.

By specifying PREAMBLE A0, UDL can also receive from SOLIS selected document information such as the DTG, long title, and the serial number. These will be interpreted as visible fields in UDL.

## TIPS/TILE

OVERVIEW – The Technical Information Processing System (TIPS) was conceived in 1960. By 1965, TIPS was operational on two Univac 490 systems, handling user requests on a 24-hour, seven-day week basis. This was a somewhat restricted system, however, handling only one request at a time and having no standard query language available for the various user data files. Presently, TIPS runs on three Univac 494 computers, where one is completely dedicated for TIPS processing. A standard query language is now available, called the TIPS Interrogation Language (TILE). Connected to the Univac 494 are 24 Honeywell 516 minicomputers. Each 516 can handle a separate query simultaneously, thus providing TIPS with simultaneity of operation. TIPS operates under the RYE operating system.

The files accessed with TILE are either single-format or multiformat files. The records within both types of files are fixed size. Within the single-format file, all records have the same format, contain the same fields, and are the same size. In a multiformat file, up to 60 formats may be used. Files, formats, and fields are called by name.

The TILE language is composed of statements. TILE statements normally begin with verbs followed by nouns specifying files, fields, and data, related by connectors and relations. The verbs available under TILE include: EXT (extract), PRINT, SORT, CREATE, POST, USE, BUILD, PUBLISH, CONST, INVOK, and UPD. PRINT has its obvious meaning. SORT allows the user to sort the output of an extract in some order other than that which is maintained by the system. CREATE and POST allow the user to create a temporary subfile. USE allows the user to leave the TILE language for special-purpose functions and return to the TILE language. BUILD allows the user to structure canned print format specifications which can be activated by the PUBLISH verb for subsequent printing. CONST gives the user the ability to construct TILE statement strings which can be activated by the INVOK verb. UPD is the update function for files.

A sample database has been established for TILE in UDL format and terminology (see figure 27). There are three records, each describing a particular employee. Each record contains information such as employee name, title, salary, spouse, birthdate, children's names, and birthplace.

## TILE DATA DESIGN –

Data Structures – There are two field structures in TILE: single-valued fields and multivalued fields which correspond to the TILE family field. These are illustrated in figure 27. CHILDNAM is a multivalued field; the others are single-valued.

The record in UDL corresponds to the record structure in TILE. UDL files correspond to TILE files, and likewise for databases.

Data Types – Field types in TILE are numeric, alphanumeric, or fixed-length text. The sample database SALARY is numeric, BIRTHPLA is fixed-length text, and the other fields are alphanumeric.

Data Attributes – All fields in the sample data base are keyed and visible. For actual TILE files, all fields will be keyed and visible. The field NAME is designated major, the others are nonmajor.

TILE INTERROGATION/DISPLAY – Figure 28 illustrates a relatively straightforward example of some mappings of operators in UDL to operators in TILE. The objective of the FIND statement is to select all records in

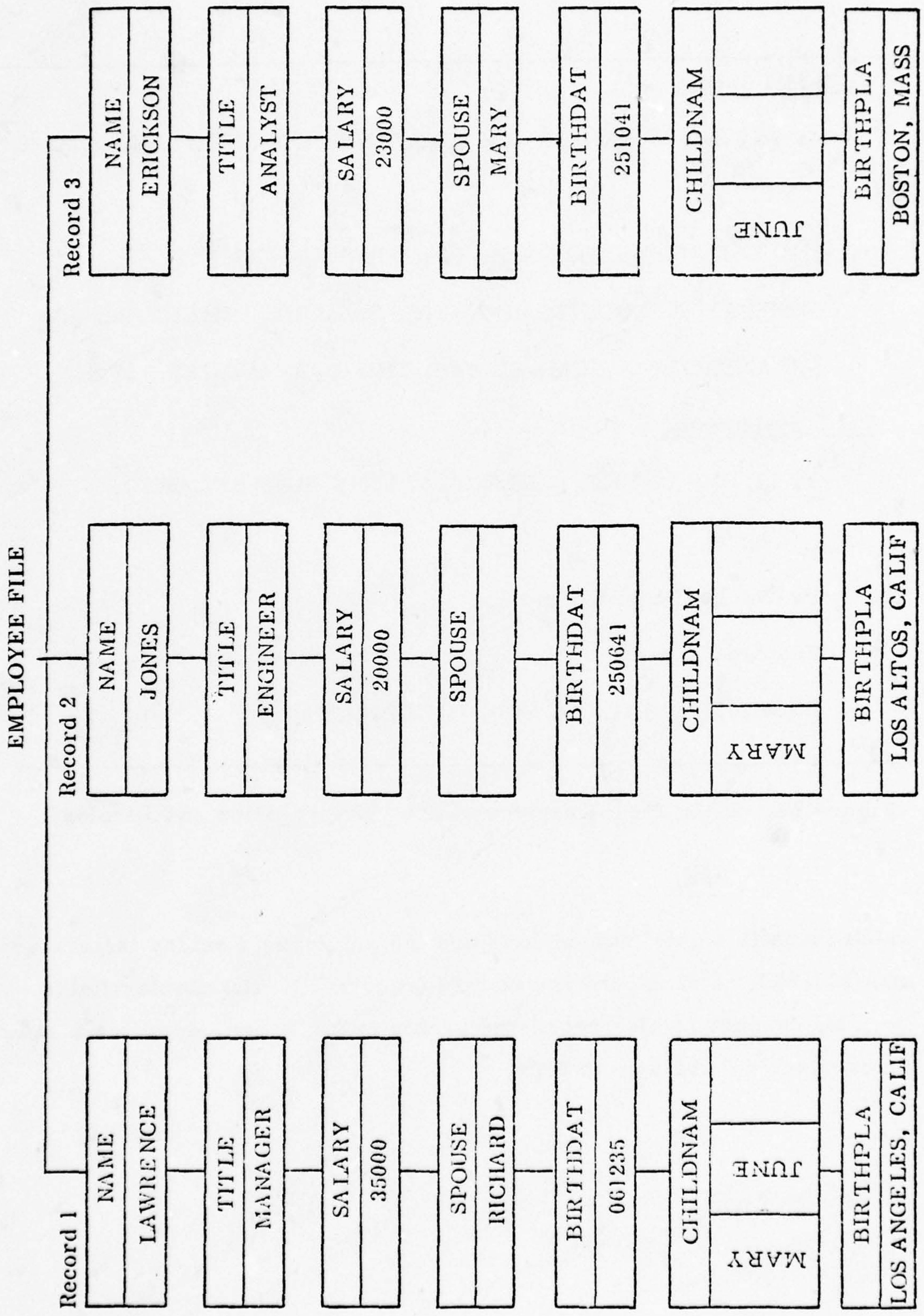


Figure 27. TILE Sample File.

```
UDL Statements

        label FIND IN EMPLOYEE CHILDNAM EQ 'JUNE' AND SALARY

        GT 23000;

        [NO HIT COUNT FOR A BATCH TRANSACTION]

        DISPLAY SOURCE (label) NAME, SALARY, CHILDNAM;

        [TRANSACTION NUMBER FOR THIS BATCH QUERY IS X]

TILE Statements

        EXTI/EMPLOYEE; CHILDNAM [JUNE] AND SALARY

        > (23000).

        PRINT EMPLOYEE:

        FORMAT: PART

        NAME (X1), SALARY (X2), CHILDNAM (X3).
```

Figure 28. UDL/TILE Transformation (Interrogation and Display).

which there is a child named June and the employee's salary is greater than $23,000. One record is selected (record 1). The display fields from all records is also requested in this example and generates a batch query which will be sent to RYE.

TILE UPDATE – The update example shown in figure 29 illustrates a change of the SALARY field from 35000 to 38000. Notice that the selection criteria had to be repeated in the TILE update command since there is no command dependency facility in TILE.

Figure 30 illustrates the removal of employees from the employee file. In this example, all employees named JONES (record 2) are to be removed. For the equivalent statement sequence in TILE, the DEL ALL subfunction of the UPD statement is utilized. Note that the file-name EMPLOYEE must be specified as well as the selection criteria. Figure 31 shows how a new employee, Gonzales, is added to the employee file. The TILE ADD sub-function is used as a transformation where each field initialization is specified.

UDL Statements

```
    label FIND IN EMPLOYEE CHILDNAM EQ 'JUNE' AND
    SALARY GT 23000;
    [NO HIT COUNT FOR A BATCH TRANSACTION]
    CHANGE SOURCE (label) SALARY TO 38000;
```

TILE Statements

```
    EXTI/EMPLOYEE; CHILDNAM [JUNE] AND SALARY > (23000).
    UPD/EMPLOYEE; SUB ALL CHILDNAM [JUNE] AND SALARY
    > (23000) TO SALARY (38000).
```

Figure 29. UDL/TILE Transformation (Update).

```
UDL Statements

    label FIND IN EMPLOYEE NAME EQ 'JONES';
    [NO HIT COUNT FOR A BATCH TRANSACTION ]
    REMOVE SOURCE (label);

TILE Statements

    EXTI/EMPLOYEE; NAME [JONES].
    UPD/EMPLOYEE; DEL ALL NAME [JONES].
```

Figure 30. UDL/TILE Transformation (Update).

```
UDL Statements

    CREATE IN EMPLOYEE NAME EQ 'GONZALES', TITLE
    EQ 'ENGINEER', SALARY EQ 25000, CHILDNAM EQ 'JOSE'

TILE Statements

    UPD/EMPLOYEE; ADD NAME [GONZALES] AND TITLE
    [ENGINEER] AND SALARY [25000] AND CHILDNAM [JOSE].
```

Figure 31. UDL/TILE Transformation (Update).

## ADAPT I ENVIRONMENT

This section presents a general picture of the ADAPT I system as it operates in a UNIX environment, and should be read before one attempts to digest the other sections describing the various ADAPT I processes and global data structures. It is also recommended that "ADAPT I Uniform Data Language (UDL): A Final Specification," 30 January 1978, be consulted before reading this section. There are many references in this section, as well as in other sections, to the many UDL/DDL/TDL statements and commands provided under ADAPT I and, therefore, a reasonable understanding of these language constructs will greatly enhance the reading of this document.

The material presented in this section has been divided into three parts: 1) a somewhat detailed description of the ADAPT I file environment, 2) a brief description of the ADAPT I System Generation Programs, and 3) a general narrative discussion illustrating the overall communication paths which are exercised by the various ADAPT I processes. After reading this section one should have a general understanding of the overall system architecture comprising ADAPT I.

ADAPT I FILE ENVIRONMENT – One design goal that has been constantly pursued was to utilize the hierarchical file structures supported under the UNIX operating system. Although many of the ADAPT I global data files are internally structured, they have been embedded in a somewhat elaborate UNIX hierarchical file structure.

The ADAPT I file environment described in the following pages is the entire data set over which all ADAPT I processes operate. This data set is composed of UNIX directories and ordinary files. Some of the directories have a fixed number of entries, while others are added to or deleted from dynamically by ADAPT I processes. Some of the ordinary files, most of which adhere to some internal structuring imposed by ADAPT I, always exist in this environment although they may change in size. Other ordinary files may or may not exist depending on what tasks ADAPT I is performing.

Figure 32 illustrates the entire ADAPT I file environment currently in use for ADAPT I. The root of the ADAPT I system file environment is attached to the "usr" directory of UNIX. The "/usr/asy" directory contains sixteen permanent entries: file pointers to eight structured global files, and pointers to eight permanent directories.

The eight permanent global files pointed to in "/usr/asy" are the user description files, GUSER (pathname "/usr/asy/guser"); the file description file, GFILD (pathname "/usr/asy/gfild"); the file name file, GFILN (pathname "/usr/asy/gfiln"); the logical transaction description file, GTDES (pathname "/usr/asy/gtdes"); the error message description file, GFEDES (pathname "/usr/asy/gfedes"); the error message text file, GFEMES (pathname) "/usr/asy/gfemes"); the tranfile file description file, GTFLD (pathname "/usr/asy/gtfld"); and the tranfile file names file, GTFLN (pathname "/usr/asy/gtfln"). GUSER contains an entry for each user registered under ADAPT I and is only increased or decreased in size if a user is added or removed. Similarly, the file global data, GFILD, GTFLD, GFILN, and GTFLN contain entries for each UDL file defined within ADAPT I (i. e., files that can be queried by ADAPT I users), their sizes changing as new UDL files are added or old files deleted. The logical transaction description file, GTDES, is more dynamic in nature, only containing entries when transactions are

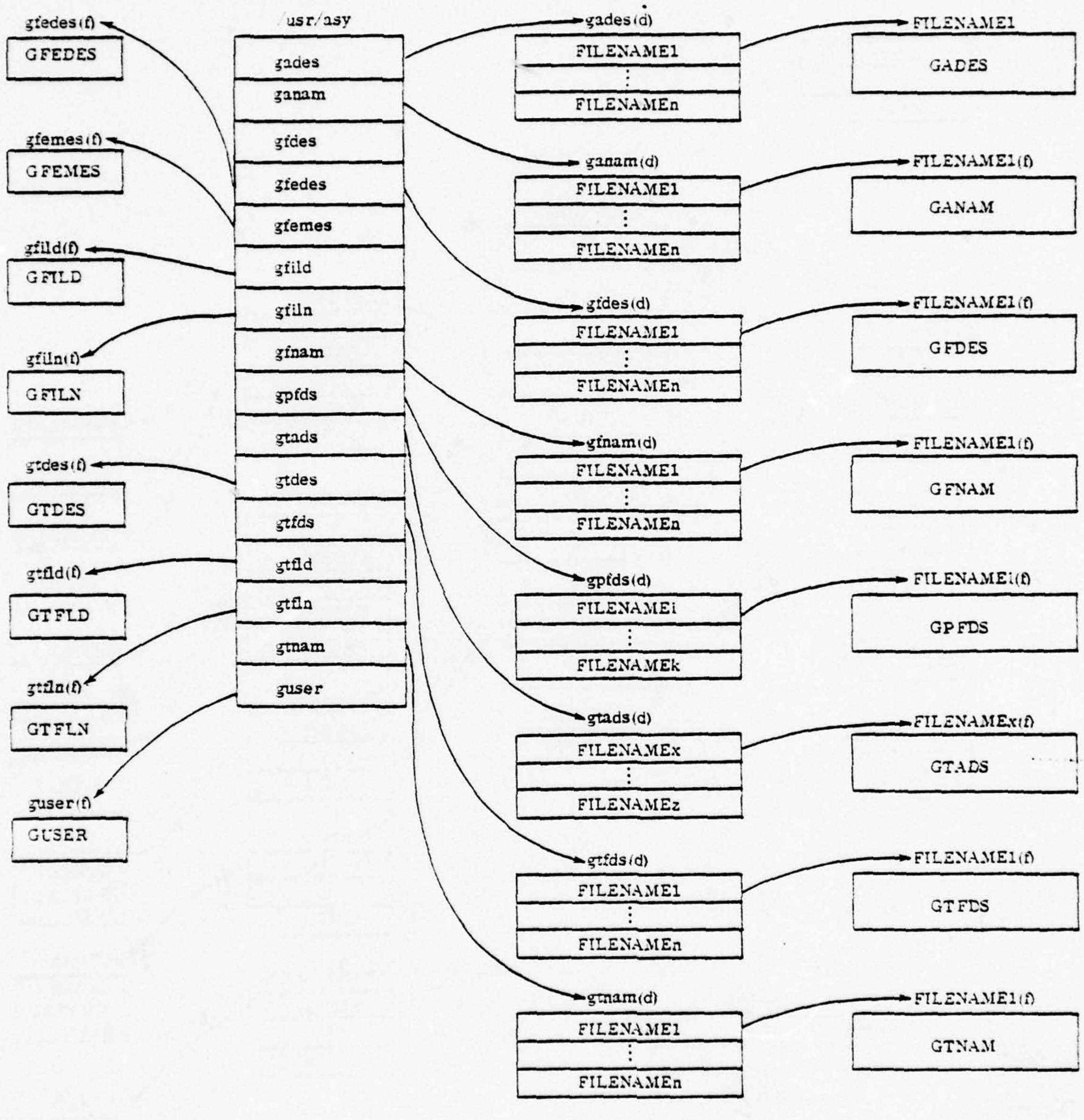


Figure 32. ADAPT I File Environment (Sheet 1 of 2)

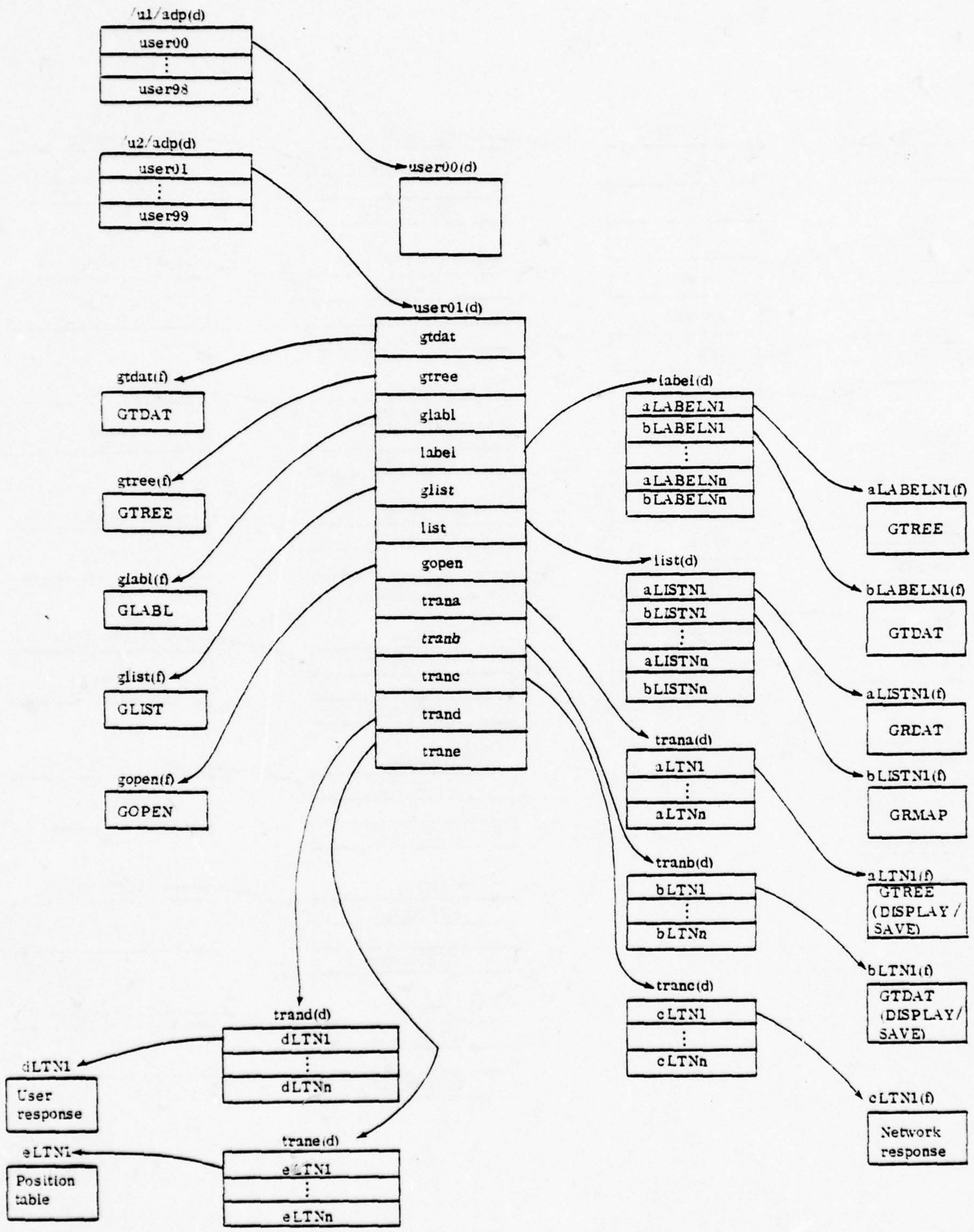


Figure 32. ADAPT I File Environment (Sheet 2 of 2)

currently active within ADAPT I; i. e. , either when a transaction is still waiting for a response from some distant batch host system, or when the response has been received but the user has not yet requested to see the output. The two files GFEDES and GFEMES are built by the ERRINIT process and are used by GFERROR to output appropriate errors and messages to a user's terminal.

As stated previously, there are eight permanent directory entries in "/usr/asy." These directory entries are used to contain pointers to the UDL file dictionaries. Their pathnames are "usr/asy/ganam," "/usr/asy/gades," "/usr/asy/gfnam," "/usr/asy/gfdes," "/usr/asy/gpfds," "/usr/asy/gtads," "/usr/asy/gtfds," and "/usr/asy/gtnam." Six of the eight directories contain a single entry for each UDL file defined under ADAPT I, while directories "/usr/asy/gpfds" and "/usr/asy/gtads" have an entry for those UDL files requiring position data and aggregate tranfile description data, respectively. Each entry in these directories is the UDL name of the file and points to a structured global file. The structure pointed to by pathname "/usr/asy/gades/filex" is the global data file GADES, the aggregate description file. Similarly, pathnames to the other dictionary files point to global structures: GANAM, the aggregate name file; GFNAM, the field name file; GFDES, the field description file; GPFDS, the field position description file; GTADS, the tranfile aggregate description file; GTFDS, the transfile field description file; and GTNAM, the tranfile aggregate and field name file. The eight global dictionary files remain fixed in size for the duration of the UDL file's existence under ADAPT I. If a file is added or deleted, entries are made to or removed from these file dictionary directories and, subsequently, new global data files are either created or deleted.

Each user registered under ADAPT I has an entry in a user directory, either "/ul/adp" or "/u2/adp". These entries are pointers to individual user directories for each registered user. They are named "/u<u>z</u>/adp/

userxy" where "xy" is an internal user ID (UID) established by ADAPT I and "z" is the algebraic remainder of the UID divided by the number of user directories (2), added to an ASCII one.

Each user directory can contain up to 12 entries: five global data file pointers and seven permanent directory pointers. Two of the global data entries point to the parse tree representation of a UDL, DDL or TDL statement or command that the user has entered from his terminal. Parse tree global data files, GTREE and GTDAT, are created/deleted each time a user enters a statement or command. Therefore, files "/uz/adp/userxy/gtree" and "/uz/adp/userxy/gtdat" do not exist unless the user (identified by a UID of "xy") is currently logged onto ADAPT I. The open files file, GOPEN, is pointed to by the pathname "/uz/adp/userxy/gopen," and contains indexes of files which the user has opened during his current session. The UDL statement label file, GLABL, is pointed to by the pathname "/uz/adp/userxy/glabl." This global data file contains the statement labels for that user during a current session. Therefore, GLABL and GOPEN are similar to the parse tree files, GTREE and GTDAT, since they only exist while a user is logged onto ADAPT I. The user list description file, GLIST, is pointed to by the user directory and has the pathname "/uz/adp/userxy/glist." Unlike the label file, this global data file exists whether the user is logged on or not. It contains descriptions of user defined lists (explicit lists) and possible implicit lists created internally by ADAPT I for active transactions. Its size may increase or decrease, depending on how many lists the user has defined.

There are seven permanent directory entries found in each individual user directory: one pointing to UDL list record data, "/uz/adp/userxy/list;" one pointing to parse tree data represented by a statement label, "/uz/adp/userxy/label;" and five pointing to information representing a logical transaction; "/uz/adp/userxy/trana," "/uz/adp/userxy/tranb," "uz/adp/userxy/tranc," "/uz/adp/userxy/trand," and "/uz/adp/userxy/trane." The user list record data directory contains two entries for each list described in GLIST. The first entry points to a global data file structured as a set of UDL record maps, GRMAP; the second entry points to a global data file structured as a set of UDL record data, GRDAT. This implies that all file records contained locally in ADAPT I are associated with a list name. The naming convention utilized to differentiate the two entries in "/u_z_/adp/user_xy_/list" is as follows. The alphabetic character "a" is concatenated to the list name as a prefix to designate GRMAP (i.e., "u_z_/adp/user_xy_/list/_a_listname") and the alphabetic character "b" is concatenated as a prefix to designate GRDAT. Implicit list names are generated internally by ADAPT I when the user requests the DISPLAY of record data from some remote file residing on a distant host system. This request can occur in two ways:

a. The remote file may reside on a batch host system and therefore a logical transaction must be initiated to that host system. This transaction is assigned a logical transaction number (LTN, $1 \leq \text{LTN} \leq 255$) unique to that user. The alphabetic character "a" concatenated to the LTN is used to form the implicit list name as it is actually inserted into GLIST and "aaLTN" and "baLTN" are the ending file designators in the pathnames for the associated record information, GRMAP and GRDAT.

b. The remote file may reside on an interactive host system. Since an ADAPT I user can only converse with one interactive system at any one time, the logical transaction number, 000, has been reserved for interactive queries. Therefore, "a000" is used as the implicit list-name and "aa000" and "ba000" are used to designate the record global data files.

The user label directory, "/u_z_/adp/user_xy_/label," is required to point to parse tree information representing the UDL statement to which the label was affixed. These global data files are GTREE and GTDAT. For each label entry in GLABL there exist two entries in the user label directory, one pointing to GTREE and the other pointing to GTDAT. The naming convention of this directory is to prefix the alphabetic "a" to the label name to designate GTREE (i.e., "/u_z_/adp/user_xy_/label/_a_labelname") and to prefix "b" to the label name to designate GTDAT.

Functionally, the original parse tree global data files that reside at "/u<u>z</u>/adp/user<u>xy</u>/gtree" and "/u<u>z</u>/adp/user<u>xy</u>/gtdat" are linked to form these entries in the user label directory. As stated earlier, when the user logs off from ADAPT I, all GLABL entries are removed and consequently so are the entries in the user label directory.

The last user directories are the user transaction directories: "/u<u>z</u>/adp/user<u>xy</u>/trana", "/u<u>z</u>/adp/user<u>xy</u>/tranb", "/u<u>z</u>/adp/user<u>xy</u>/tranc", "/u<u>z</u>/adp/user<u>xy</u>/trand", and "/u<u>z</u>/adp/user<u>xy</u>/trane". These directories are used primarily for controlling the batch transactions which a user initiates. Each transaction directory contains an entry for each logical transaction currently active for a user; optional parse tree data, GTREE and GTDAT, representing a DISPLAY statement; a transformation string file which also doubles as a response string repository; a temporary output file to contain the final user output prior to his perusal; and a position data file containing data positions for specific fields in the response string. The naming conventions utilized for these five directory entries are the concatenation of the alphabetic characters "a," "b," "c," "d," and "e," to the logical tranaction number: "aLTN" designates GTREE, "bLTN" designates GTDAT, "cLTN" designates the transformation/response string file, "dLTN" designates the temporary output file and "eLTN" designates the position data file.

<u>The ADAPT I SYSTEM GENERATION (SYSGEN) PROGRAMS</u> – Due to the somewhat complicated UNIX file environment utilized by ADAPT I, it has been necessary to provide interactive Sysgen Programs which will generate this hierarchical file structure. The initial Sysgen Program must be initiated by the UNIX superuser since directories are being created as well as deleted. The ADAPT I Sysgen Programs perform two functions:

a. Initial mode – creates the baseline UNIX file structure.

b. Update mode – allows the registration and/or deletion of ADAPT I users.

The initial mode program, called ADINIT, will establish the following permanent directories and files for the ADAPT I environment.

a. /usr/asy/(directory).
b. /usr/asy/guser (file).
c. /usr/asy/gfiln (file).
d. /usr/asy/gfild (file).
e. /usr/asy/gtdes (file).
f. /usr/asy/gtfld (file).
g. /usr/asy/gtfln (file).
h. /usr/asy/ganam (directory).
i. /usr/asy/gades (directory).
j. /usr/asy/gfnam (directory).
k. /usr/asy/gfdes (directory).
l. /usr/asy/gpfds (directory).
m. /usr/asy/gtads (directory).
n. /usr/asy/gtfds (directory).
o. /usr/asy/gtnam (directory).
p. /ul/adp (directory).
q. /u2/adp (directory).

The directories "/usr", "/ul" and "/u2" are assumed to exist. Upon the generation of baseline UNIX files and directories, the ADINIT Program creates the ADAPT I superuser. The following directories and files are created for the ADAPT I superuser:

a. /ul/adp/user00 (directory).
b. /ul/adp/user00/trana (directory).
c. /ul/adp/user00/tranb (directory).
d. /ul/adp/user00/tranc (directory).

e. /ul/adp/user00/trand (directory).

f. /ul/adp/user00/trane (directory).

g. /ul/adp/user00/glist (file).

h. /ul/adp/user00/list (directory).

i. /ul/adp/user00/label (directory).

The second ADAPT I Sysgen Program is a command which can only be utilized by the TASMASTER when logged onto TAS. This command, called ADUSER, interacts with the TAS superuser and prompts him to enter or delete the next ADAPT I user. At this point, new ADAPT I users (who must be unique within ADAPT I) can be registered or deleted, the current users can be listed, or the program can be terminated. For each new user registered, the directories and files which were created for the superuser are created for that user's ID.

As implied by the foregoing discussion, the ADAPT I system requires a superuser. This user has special privileges not available to other ADAPT I users. For example, only the ADAPT I superuser can define a file's SCHEMA and TRANFILE or delete a file's SCHEMA and/or TRANFILE. The ADAPT I superuser is the same individual who is the TASMASTER.

ADAPT I PROCESSES – The term "process" is used herein in the literal sense as in UNIX; i. e., an executable file (program) which can be initiated either through terminal commands entered by a user who has execute access rights, or by another executing process via the execute system call.

Figure 33 illustrates seventeen processes currently utilized for ADAPT I, showing their communication paths with each other, with the user at the terminal, and with the COINS II Network. Also provided in figure 33 is some general association of the UDL/DDL/TDL statement/command set with the process(es) which perform the appropriate operation.

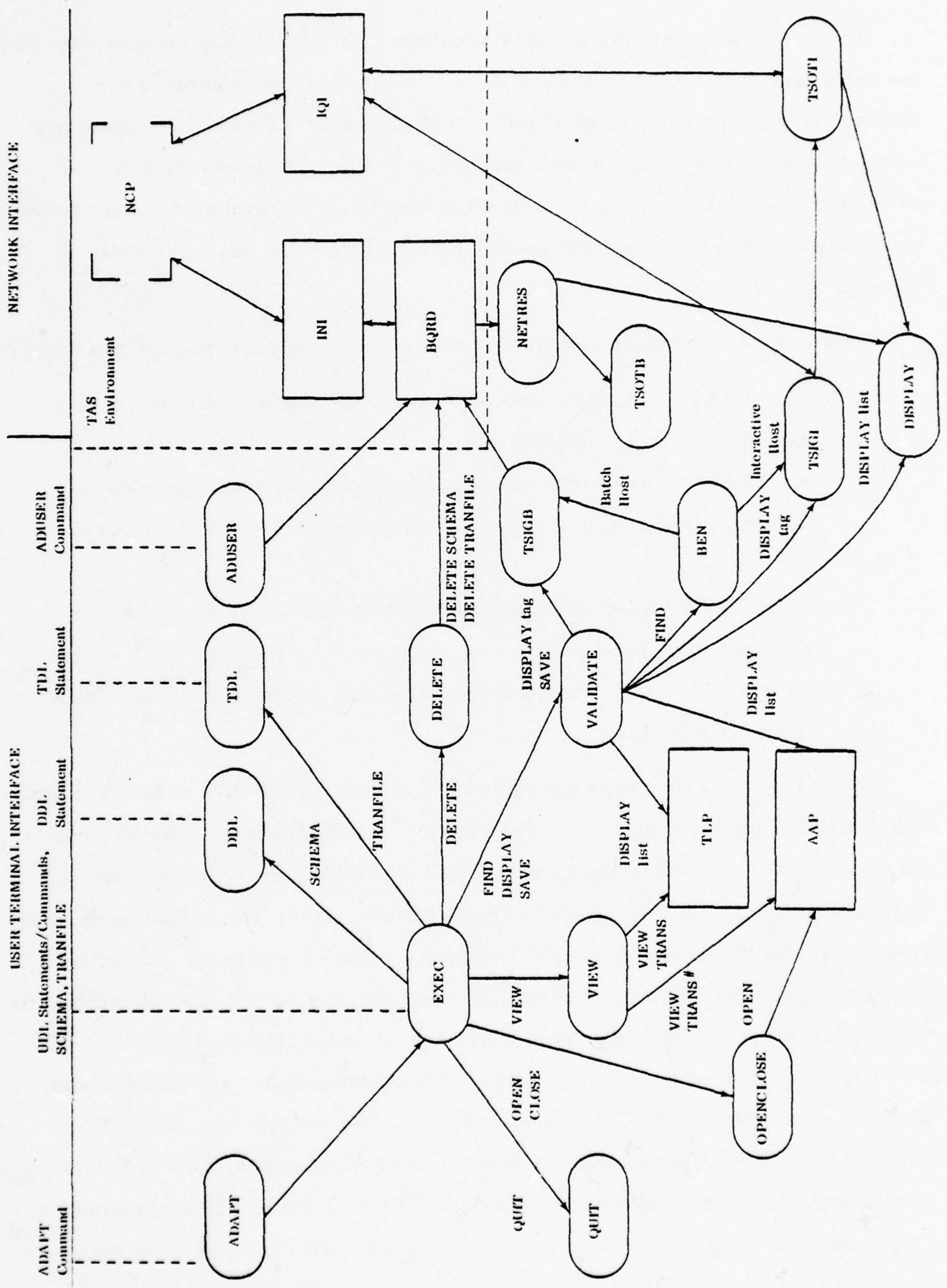


Figure 33. ADAPT I Process Communication

Probably the best way to illustrate the interaction that occurs between the different ADAPT I processes is to present a set of simple user scenarios involving user statement/command entries and corresponding network responses. The scenarios presented in the following are not necessarily complete with respect to actual user statement entries; however, they do provide a general framework in which to view the operation of ADAPT I.

The discussion which follows is concerned with four simple scenarios:

a. The ADAPT I superuser calls up ADAPT and describes a new file via the DDL sublanguage.

b. An ADAPT I user (not necessarily the superuser) queries an interactive file and displays a set of fields from the selected record set.

c. An ADAPT I user queries a batch file and saves the selected records in a list.

d. An ADAPT I user views a specified file schema and then exits from ADAPT I.

<u>Scenario 1</u> – For users (any user) to utilize ADAPT I, they must enter the ADAPT command through a terminal. The TAS Shell, which recognizes this command, activates the process called ADAPT which performs user validation. If the user is an authorized ADAPT user, hence has been registered by the ADAPT I ADUSER Program, ADAPT activates the ADAPT I executive process, EXEC. From this point on all user/terminal interfaces are either directed entirely from EXEC or at least initiated from EXEC. The user can now proceed with other UDL commands or statements and, if he or she is the ADAPT I superuser, can also utilize DDL and TDL. The EXEC process performs the front-end processing of all UDL statements and commands as well as the SCHEMA and TRANFILE statements. Depending on the statement or command, EXEC may process the statement itself or call one or more other processes for processing.

Assume that the ADAPT I superuser has entered ADAPT and types a SCHEMA statement. Since this is the superuser, EXEC calls the DDL process for processing of the schema definition. The DDL process is responsible for generating UDL file dictionaries representing the logical structure of the specified file. These dictionaries are utilized by most of the other ADAPT I processes during their processing of various user statements and commands. DDL has its own front-end which is constructed to recognize DDL statements; i. e., field definitions, aggregate definitions, etc. From this point on, DDL has control of the user's terminal and does not return to the EXEC process until the user enters an ESCHEMA statement. Since a user may have a text file of DDL statements processed (via the EXECUTE command), the EXEC process, which processes the EXECUTE command, establishes the user specified text file as the system input prior to calling DDL. Upon completion of the DDL process, whether DDL has been interacting with the user or reading from a user's text file, control is returned to the EXEC process.

Scenario 2 – Assume at this point a user has successfully entered ADAPT and is now interacting with the EXEC process. The user now wishes to query file-x which happens to reside on an interactive distant host. The front-end of EXEC checks the syntactic validity of the entered FIND statement and, if the statement is valid syntactically, calls the VALIDATE process for semantic validation. VALIDATE performs two basic functions:

a. It verifies that the user's statement, although syntactically correct, is also compatible with the file being referenced. This involves verifying that the proper relational operators are being used against the correct data types, the user is not violating the data structures' access attributes, and the data structures as named do indeed exist for the specified file.

b. The internal representation of the user's statement (a parse tree) is modified to reflect more discerning information than was possible during the parsing process performed by the EXEC front-end.

Upon successful completion of the VALIDATE process, VALIDATE calls the Boolean Expression Normalizer (BEN) which normalizes and minimizes the selection criteria in the FIND statement. After this has been done, BEN calls the Target System Input Generator (TSIGI, the interactive version) for the actual execution of the FIND statement. TSIGI is responsible for performing the transformation of UDL statements into the appropriate distant host query language. TSIGI creates the proper entry in the label table (GLABL) for the FIND statement and saves the modified parse tree and parse tree data files by linking them into the label directory. Transformation strings are generated by TSIGI and are sent through the network indirectly via a pipe write to the TAS Interactive Query Interface (IQI). TSIGI then waits for a response (assuming the distant host is indeed available and the initial connection was successful). Normally the response will be the distant host's acknowledgement of the query and the number of records satisfying it. This information is output to the user as a UDL record hit count. TSIGI then returns to EXEC which, in turn, prompts the user for his next statement or command. Although in practice a user will normally want to display some portion of the records he has selected, he may also wish to query another file (which may exist on an entirely different distant host system). File/host system association is known by ADAPT I and, therefore, the user does not have to be concerned with the actual system the file resides on.

Assume now that the user is satisfied with the results of the FIND statement just entered and desires to display a set of fields from the selected record set. This is accomplished by entering a DISPLAY statement which references a previously input FIND statement (by a statement label). Upon successful entry of the DISPLAY statement into EXEC, the VALIDATE process is again called to check the semantic validity of the user's statement. Assuming there were no errors, the VALIDATE process calls TSIGI for further processing. As was the case with the FIND statement, TSIGI generates the appropriate transformation strings and transfers them across the network to the respective distant host. TSIGI then calls the Target System Output Translator (TSOT) process for analysis of the host responses. TSOT is responsible for translating the various host system record outputs into compatible UDL internal record formats.

TSOT has the difficult task of distinguishing host error messages from host record data and converting the host data into a set of UDL internal records. If an error message is encountered from the host system, TSOT translates the error message into a consistent UDL error message and outputs it to the user. After building the first UDL internal record, TSOT calls the DISPLAY process to process the user's DISPLAY statement. At this point, the peculiarities of the different distant host systems have disappeared entirely and the DISPLAY process expects and processes only data that are in the UDL record format. DISPLAY extracts the data as specified in the DISPLAY statement, outputs these data in UDL default format, and after processing all records, exits.

Scenario 3 – In the previous scenario, the user interrogated a file that happened to exist on an interactive host system. In this scenario, the user interrogates a file which resides on a batch host system. These distinctions are brought out to illustrate the inherent differences in processing techniques that ADAPT I must go through when an interactive file is queried as compared to a batch file.

To accommodate these differences, different TSIG processes exist in ADAPT I; TSIGI for interactive file references and TSIGB for batch file references.

Operationally within ADAPT I, when an interactive file is referenced, a connection is established with the distant interactive host system. The ADAPT I system can be thought of as a bidirectional filter absorbing UDL as user input and generating UDL displays as output. For batch files, the user UDL statements are composed into logical transactions which are then sent to the respective batch systems for processing. The responses which eventually result from the transactions are processed and saved for the user (under logical transaction numbers) for later perusal.

Assume as in the previous scenario that a user has successfully entered a FIND statement referencing some file which resides on a batch system. Also assume that the statement has passed semantic validation. At this point, VALIDATE calls BEN to normalize and minimize the boolean expression as specified in the FIND statement. After this has been done, BEN calls the Target System Input Generator for batch file references (TSIGB process). TSIGB creates the proper entry in the label table (GLABL) for the FIND statement and saves the modified parse tree and parse tree data files by linking them into the label directory. It then returns to EXEC. Note that there was no network communication at this time. For a logical transaction to be complete, it must do more than just interrogate a file. It must ask for some data contingent upon the interrogation as well. This can be accomplished in UDL either by requesting that the selected records be saved in a local list or by displaying some portion of the record set. Notice that there is no true interaction with the user and distant host system. The user must package his transaction as described in the foregoing and then wait for the results. Upon successful entry of a SAVE command, VALIDATE semantically validates that the SAVE command references a previous FIND statement and, if it is a valid command, VALIDATE calls TSIGB for further processing. TSIGB reserves a logical transaction number for the transaction and then generates the appropriate transformation strings representing the FIND statement for that host system. TSIGB then performs the appropriate transformation on the SAVE command and generates a set of transformation strings which are added to the end of the previous strings generated by the previous FIND statement. At this point, a logical transaction can be packaged and sent through the network. To do this, TSIGB must call the TAS Batch Query and Response Dispatcher (BQRD) process, which will generate the proper COINS I INTG message and transmit it through the network via the NCP. TSIGB then outputs the logical transaction number to the user and returns to the executive, EXEC. At this point, the user is prompted and can now enter new statements or commands.

Upon receipt of a COINS I ANSR or ABRT message pertaining to the logical transaction initiated in the foregoing, BQRD will call the Network Response process (NETRES) of ADAPT I for processing. At the time TSIGB established the logical transaction through BQRD, it also specified that NETRES be called when a response for this transaction occurred.

NETRES analyzes the message received from the distant host. If the response was an ANSR, which would normally be the case, TSOT is called to perform its function of translating host system record data into proper UDL records. Upon the completion of TSOT, NETRES determines whether the user requested the display or a save of data. In this particular scenario, the user is only interested in saving the selected records in an internal list local to him. Therefore, NETRES indicates that the logical transaction is now complete and exits. If the user had requested the display of data, NETRES would have called the DISPLAY process which would display the data in UDL default format into a temporary output file associated with this particular logical transaction. At a later time, the user could peruse this output by requesting it through the VIEW command.

Scenario 4 – If a user desired to see the logical structure of some file, he may do so by entering the VIEW SCHEMA command. The EXEC process calls the VIEW process to verify the semantic correctness of the command and to output the file's logical structure. VIEW not only displays file structures, but can also display the names of files currently defined under ADAPT I (VIEW FILE); the names of lists defined locally by the requesting user (VIEW LIST); the labels used during the current session (VIEW LABEL); all batch transactions initiated by the user (VIEW TRANS), and transformation file structures (VIEW TRANFILE).

If users wish to exit from ADAPT I, they must enter the QUIT command. EXEC calls the QUIT process which removes all local statement labels existing for that user, indicates that the user is inactive, and returns control to the TAS Shell.

## ADAPT I PROCESS DESCRIPTIONS

This section provides a detailed description of the processes currently existing in ADAPT I. The processes described in this section are as follows:

a. ADAPT – processes initial user entry to ADAPT I.

b. ADINIT – generates the hierarchical file structure for ADAPT I system.

c. ADUSER – establishes the superuser and additional ADAPT users.

d. BEN – normalizes and minimizes boolean expressions in FIND statements.

e. DDL – processes all DDL statements and generates file dictionary data files.

f. DELETE – processes the DELETE command.

g. DISPLAY – processes the DISPLAY statement, displaying record data in UDL default format.

h. EXEC – performs overall executive operations for ADAPT I including UDL statement/command syntactic analysis, and the transferring of control to appropriate ADAPT I processes for statement/command processing.

i. NETRES – processes batch host system responses and maintains the user transaction data file.

j. OPENCLOSE – processes the OPEN and CLOSE UDL commands.

k. QUIT – removes the user from ADAPT I.

l. TDL – processes all TDL statements and generates files containing transformation relevant data.

m. TSIGB – generates transformation strings for queries of files residing on batch host systems.

n. TSIGI – generates transformation strings for queries of files residing on interactive host systems.

o. TSOT – translates distant host system responses into UDL internal data records.

p. VALIDATE – performs semantic validation of three UDL statements: FIND, SAVE and DISPLAY.

q. VIEW – performs all operations involving the display [illegible] file names, SCHEMA structures, user list names, us[illegible] label names, TRANFILE information and user transac[illegible]ons.

Each process description is organized as follows: A short general description is provided, followed by a list of the ADAPT I global data used. Then, if applicable, a list of pertinent local data contained in the process is provided. A general data flow is then given which narrates the overall paths the process must take in order to accomplish its tasks. After the general flow, salient functions contained in the process are highlighted. These functions, which were called out by name in the general flow description, are described in more detail. Following the process description are flow charts which illustrate the upper level program paths of the process and its salient functions.

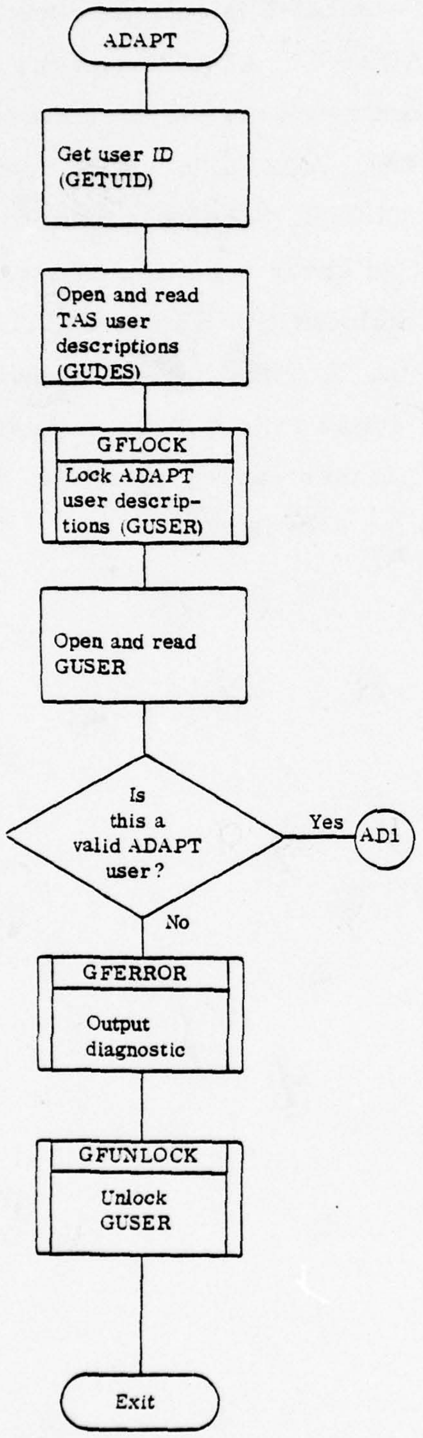
## ADAPT (ADAPT Logon Process)

The ADAPT process is activated when a user enters the ADAPT command. ADAPT validates the user's name and activates the ADAPT executive, EXEC.

GLOBAL DATA USAGE – ADAPT utilizes GUSER – user descriptions.

LOCAL DATA USAGE – ADAPT has no pertinent local data.

GENERAL PROCESS FLOW – ADAPT is called by the TAS Shell program when a user types in "ADAPT." ADAPT gets the user ID using the UNIX function GETUID and then retrieves the user name from the TAS user descriptions file (GUDES). ADAPT locks the user descriptions (GUSER) via GFLOCK and reads GUSER. The user name is validated to be a valid ADAPT user name. If an error condition exists, a diagnostic is output via GFERROR, ADAPT unlocks GUSER via GFUNLOCK and terminates, and the user is back at the TAS Shell program level. If no error is found, ADAPT sets the user's status to logged-on, writes GUSER, and unlocks GUSER via GFUNLOCK. It then passes control to the ADAPT executive, EXEC. See figure 34 for data flow.


ADAPT
Get user ID (GETUID)
Open and read TAS user descriptions (GUDES)
GFLOCK
Lock ADAPT user descrip-tions (GUSER)
Open and read GUSER
Is this a valid ADAPT user?
Yes
AD1
No
GFERROR
Output diagnostic
GFUNLOCK
Unlock GUSER
Exit

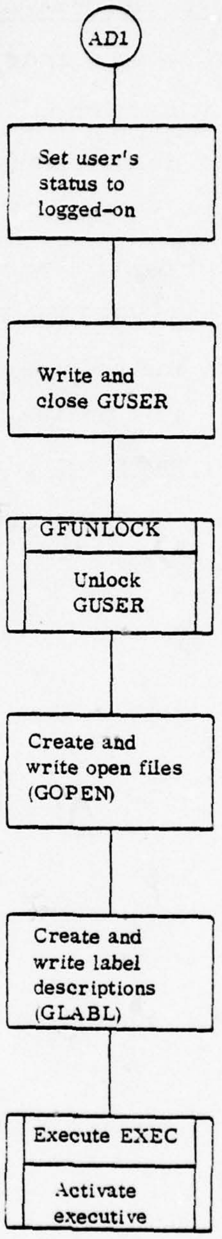

AD1
Set user's status to logged-on
Write and close GUSER
GFUNLOCK
Unlock GUSER
Create and write open files (GOPEN)
Create and write label descriptions (GLABL)
Execute EXEC
Activate executive

Figure 34. ADAPT Process Data Flow

## ADINIT (ADAPT Environment Generator)

The ADINIT process generates the fundamental directory and file environment for the ADAPT I system.

GLOBAL DATA USAGE – ADINIT utilizes the following global data files:

| | | |
|---|---|---|
| GFILD | – | file descriptions. |
| GFILN | – | file names. |
| GTDES | – | transaction descriptions. |
| GTFLD | – | transformation file descriptions. |
| GTFLN | – | transformation file names. |
| GUSER | – | user descriptions. |

LOCAL DATA USAGE – ADINIT has no pertinent local data.

GENERAL PROCESS FLOW – The ADINIT process generates the basic hierarchical directory and file structure required by the ADAPT I system. The ADINIT process validates that the superuser is the person who is calling this process. Once this has been verified, ADINIT makes the basic directory for the ADAPT system and then the user super-directories. Along with these directories, the following eight directories are made:

a. aggregate name directory (for GANAM files).

b. aggregate description directory (for GADES files).

c. field name directory (for GFNAM files).

d. field description directory (for GFDES files).

e. field position description directory (for GPFDS files).

f. transformation aggregate description directory (for GTADS files).

g. transformation field description directory (for GTFDS files).

h. transformation field and aggregate name directory (for GTNAM files).

The ADINIT process creates and writes the following files.

| | | | |
|---|---|---|---|
| a. | GUSER | – | user descriptions. |
| b. | GFILN | – | file names. |
| c. | GFILD | – | file descriptions. |
| d. | GTFLN | – | transformation file names. |
| e. | GTFLD | – | transformation file descriptions. |
| f. | GTDES | – | transaction descriptions. |

ADINIT calls the ERRINIT process which builds the two ADAPT error/message files: GFEDES (error/message descriptions) and GFEMES (error/message text). These files are to be used by the global function GFERROR.

Lastly, the ADINIT process executes the ADUSER process, the mechanism by which the superuser and additional ADAPT users are established. See figure 35 for data flow.

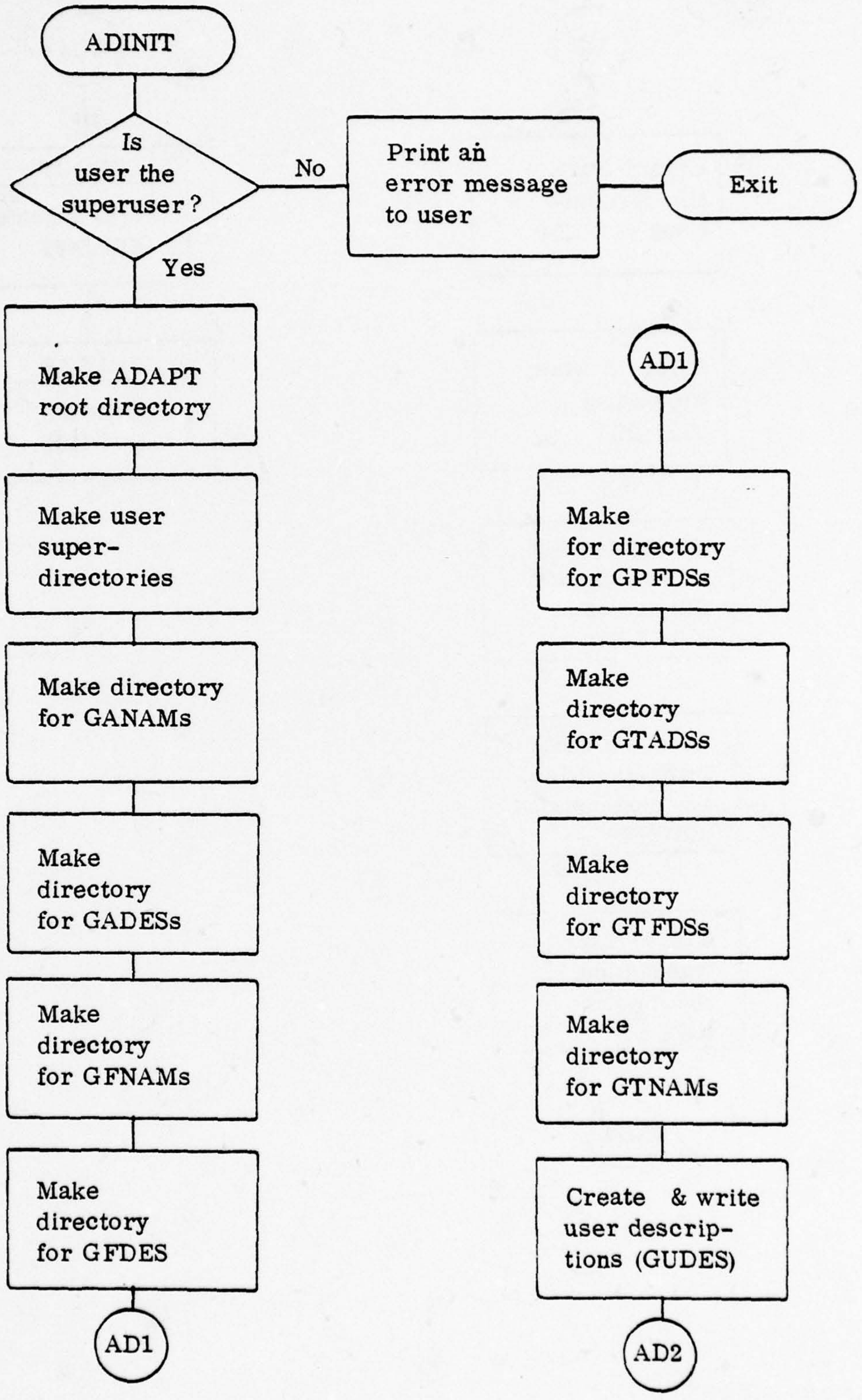

ADINIT
Is user the superuser ?
No
Yes
Print an error message to user
Exit
Make ADAPT root directory
Make user super-directories
Make directory for GANAMs
Make directory for GADESs
Make directory for GFNAMs
Make directory for GFDES
AD1
AD1
Make for directory for GPFDSs
Make directory for GTADSs
Make directory for GT FDSs
Make directory for GTNAMs
Create & write user descriptions (GUDES)
AD2

Figure 35. ADINIT Process Data Flow (Sheet 1 of 2)

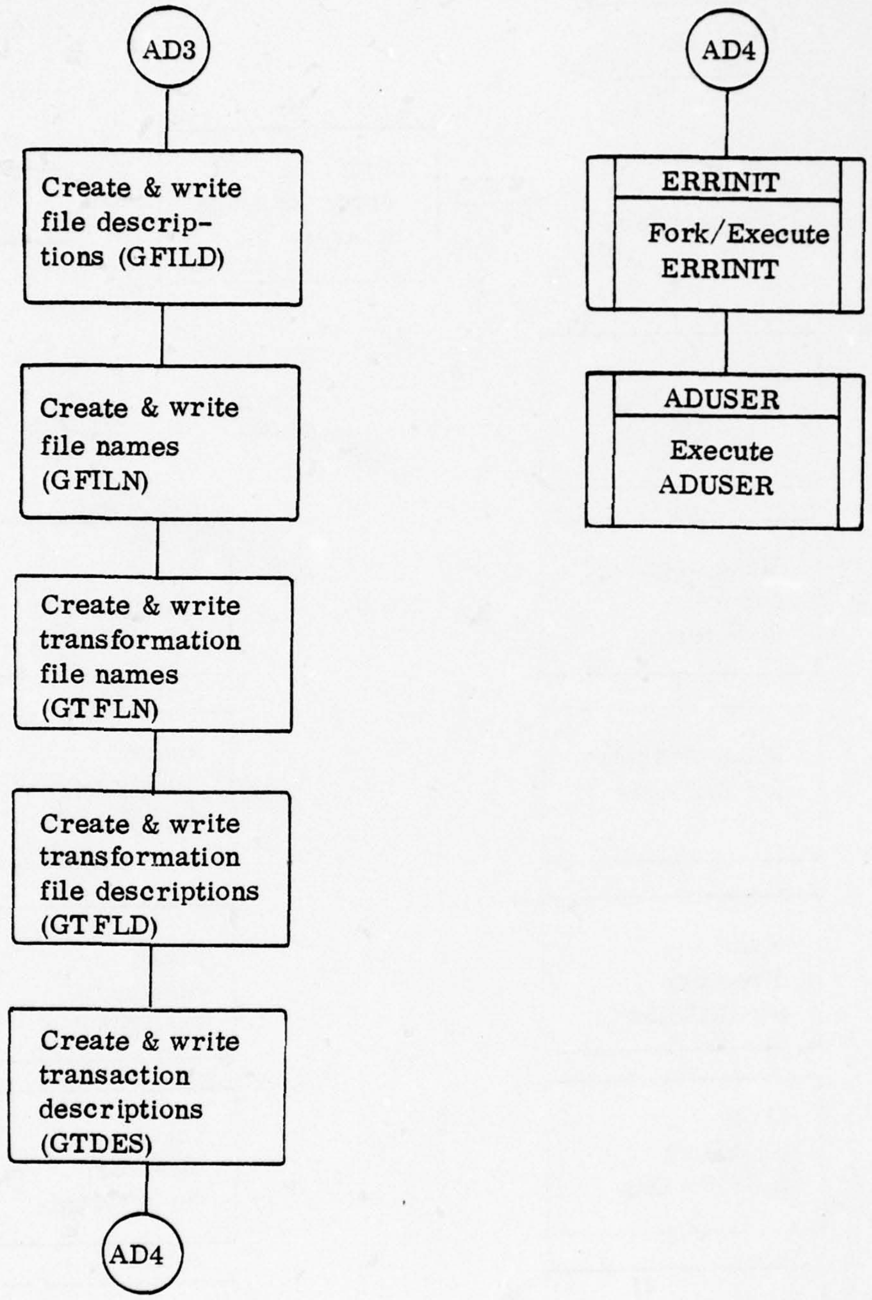

AD3
Create & write file descrip- tions (GFILD)
Create & write file names (GFILN)
Create & write transformation file names (GTFLN)
Create & write transformation file descriptions (GTFLD)
Create & write transaction descriptions (GTDES)
AD4
AD4
ERRINIT
Fork/Execute ERRINIT
ADUSER
Execute ADUSER

Figure 35. ADINIT Process Data Flow (Sheet 2 of 2)

## ADUSER (ADAPT User Maintenance)

The ADUSER process enables the TASMASTER to add ADAPT users, delete ADAPT users, or list current ADAPT users. ADUSER is a TAS level command and neither calls nor is called by other ADAPT processes except for ADINIT when the ADAPT environment is created.

GLOBAL DATA USAGE – ADUSER uses the following global data:

| | | |
|---|---|---|
| GUSER | – | user descriptions. |
| GLIST | – | list names and descriptions. |
| GTFLD | – | transformation file descriptions. |
| GTDES | – | transaction descriptions. |
| GUDES | – | TAS user descriptions. |

LOCAL DATA USAGE – ADUSER has no pertinent local data.

GENERAL PROCESS FLOW – ADUSER can be called in one of two ways: either by the ADAPT Environment Generator (ADINIT) in order to create the ADAPT superuser; or by the TASMASTER as a TAS level command. If the user is neither the TASMASTER nor the UNIX superuser, then an error is output via GFERROR and ADUSER exits. Otherwise, ADUSER locks the User Descriptions (GUSER) file via GFLOCK, and then opens and reads the first item of GUSER. If there is no ADAPT superuser defined, then a message is output requesting the superuser's name and processing continues as if an Enter User command had been input. If the ADAPT superuser is already defined, then a message is output listing the possible commands and requesting a command input. A single character is read specifying the requested command: E – Enter a User, D – Delete a User, L – List Users, or newline – Quit.

If the request is Enter User, then ADUSER checks to see if there is room in GUSER for another user. If not, an error message is output and ADUSER outputs the command request line again. Otherwise ADUSER outputs a message requesting the user's name. At this point, whether the superuser or an ordinary user is being defined, USREDNM is called to read and validate the user's name. If the superuser is being defined and there was an error reading the name, then ADUSER repeats the request for the superuser's name as above. If any other user is being defined, and there was an error reading the name, then the command request line is output again. If the user name is already defined in ADAPT, then an error message is output and ADUSER outputs the command request line again. A final validation is made by USTAS to ensure that the name is a valid TAS name (or SNI). If not, then either the command request line or the request for the superuser's name is repeated. After all validation tests against the name are completed satisfactorily, the name is stored in an empty item in GUSER. The following directories are then created for the user: basic directory, list directory, label directory and five transaction directories. The user's List Names and Descriptions (GLIST) file is then created and written out. With the completion of this user's definition, ADUSER outputs the command request line.

If the request is List Users, then ADUSER first outputs a header. The user's name and current ADAPT status (logged or not logged onto ADAPT) are output for each user defined in ADAPT. After all users have been processed, ADUSER repeats the command request line.

If the request is Delete User, then ADUSER outputs a message requesting the user's name. USREDNM is then called to read and validate the name. If there was an error reading the name, then ADUSER outputs the command request line again. If the name is not defined under ADAPT, if the name is the superuser's name, or if the specified user is currently logged onto ADAPT, then an error message is output and the command request line is repeated. Otherwise, the number of transactions for the user is picked up from GUSER and the user is then deleted from GUSER. The Transformation File Descriptions (GTFLD) file is opened and item zero is read. The Transaction Descriptions (GTDES) file is locked via GFLOCK. ADUSER then opens and reads the user's GLIST file. If there are any lists, then, for each list, USDELST is called to delete the list. The GLIST file is then unlinked and the list directory is deleted. ADUSER next checks to see if the user has any transactions. If the user has transactions, then the following processing is performed.

ADUSER opens and reads the first item of GTDES. Each utilized item of GTDES is read, and for each transaction belonging to the user, USBQRD is called to cancel the transaction's jobs and USDTRAN is called to delete the transaction.

ADUSER unlocks GTDES via GFUNLOCK and deletes the user's transactions directories. The user's label directory and basic directory are then deleted. After having deleted the user, ADUSER repeats the command request line.

If the request is Quit, then ADUSER writes GUSER back to disk and exits via USERROR. See figure 36 for process flow.

## MAJOR FUNCTION DESCRIPTIONS

USTAS (Validate a User Name) – USTAS opens and reads the TAS User Descriptions (GUDES) file. GUDES is searched to determine whether the input name is a valid TAS SNI. If it is, then GUDES is closed and USTAS returns with a value of OK. If the name is invalid, then GUDES is closed, an error message is output and USTAS returns with a value of error.

USREDNM (Read and Verify a Name) – USREDNM reads a user's name from the terminal. If the name has more than eight characters, then an error message is output and USREDNM returns with a value of error. If the superuser is being defined, then USREDNM returns a value of zero (superuser's user ID). Otherwise, USREDNM searches GUSER to determine whether the input name is a defined ADAPT user name. If not, USREDNM returns a value of no match. Otherwise, a value of match (matched user ID) is returned.

USDELST (Delete a List) – USDELST deletes the specified list's description from GLIST. The list's Record Map (GRMAP) and Record Data (GRDAT) files are then unlinked. USDELST opens GTDES and reads items until the list's transaction is found. If the list's data had not been displayed, then USBQRD is called to cancel the transaction's jobs. USDTRAN is called to delete the transaction, GTDES is closed and USDELST returns.

USDTRAN (Delete a Transaction) – USDTRAN unlinks the five possible transaction files: Parse Tree (GTREE) and Parse Tree Data (GTDAT) for the DISPLAY or SAVE statement, Field Position Descriptions (GPFDS) for position-oriented network responses, transaction output, and network response. The transaction description item in GTDES is set to zeroes and written out to disk. If the transaction consisted of multiple queries (multi-INTGs), then each item in GTDES which related to the transaction is also set to zeroes and written out. The first item of GTDES, which contains the total number of transactions in ADAPT, is read, the transaction count is decremented, and the GTDES item is written back out. USDTRAN then returns.

<u>USBQRD (Cancel a Transaction's Jobs)</u> – USBQRD reads the GTFLD data for the file referenced by the transaction. The application ID is picked up from GTFLD to be sent to the TAS Batch Query and Response Dispatcher (BQRD). USBQRD performs a fork/execute sequence in order to call BQRD. BQRD is called with a function code of CANCEL, the job's application ID, and the JOBID of the batch query (INTG) to be cancelled. If the transaction consisted of multiple queries (multi-INTGs), then BQRD is repeatedly called for each batch query in the transaction. After all batch queries relating to the transaction have been cancelled, then USBQRD returns.

<u>USERROR (Output an Error and/or Exit)</u> – USERROR is called with an error number or zero. If an error number is specified, then the specified error message is output via GFERROR. If GTDES is locked, then GFUNLOCK is called to unlock it. USERROR then unlocks GUSER via GFUNLOCK and exits.

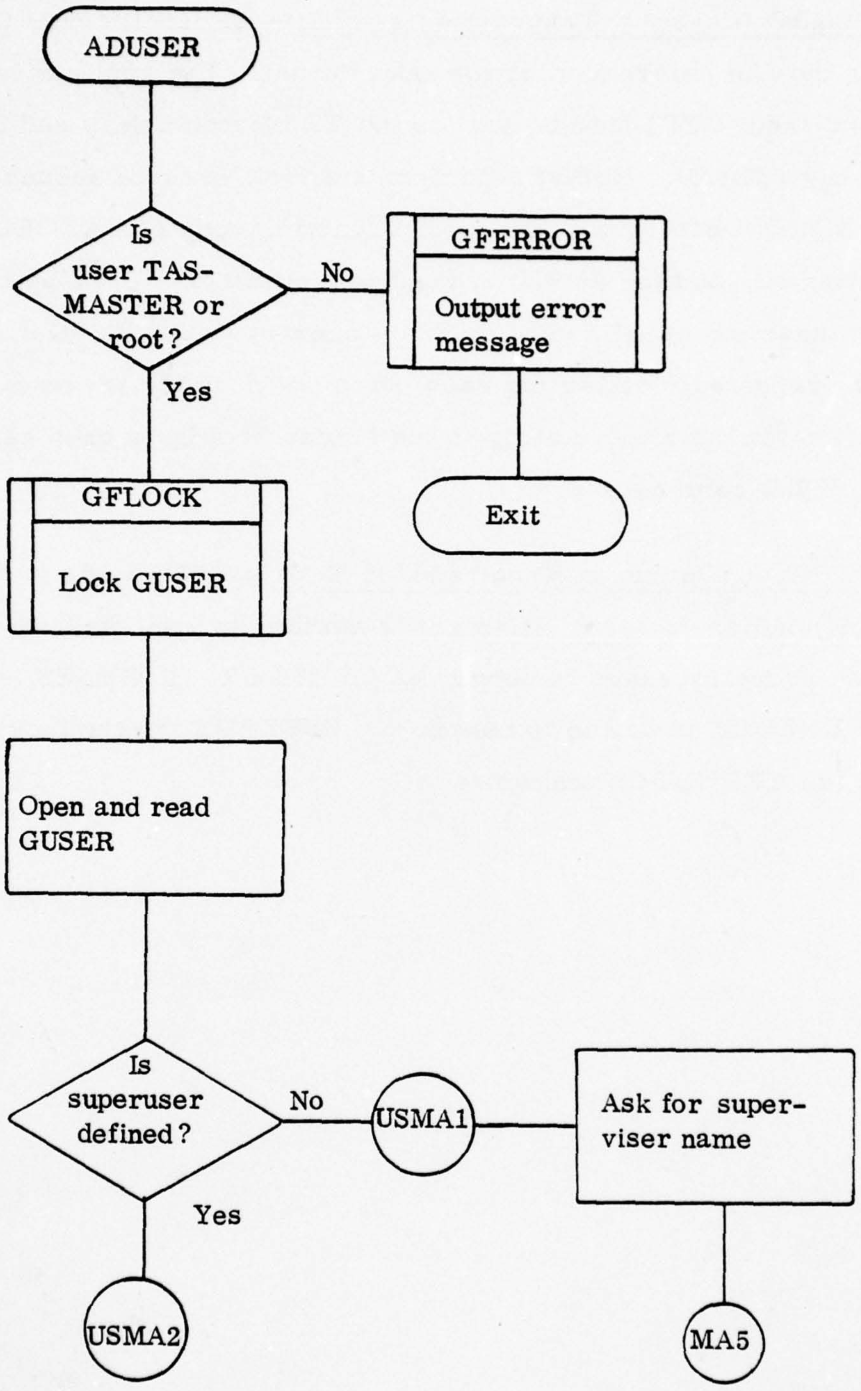


Figure 36. ADUSER Process Data Flow (Sheet 1 of 12)

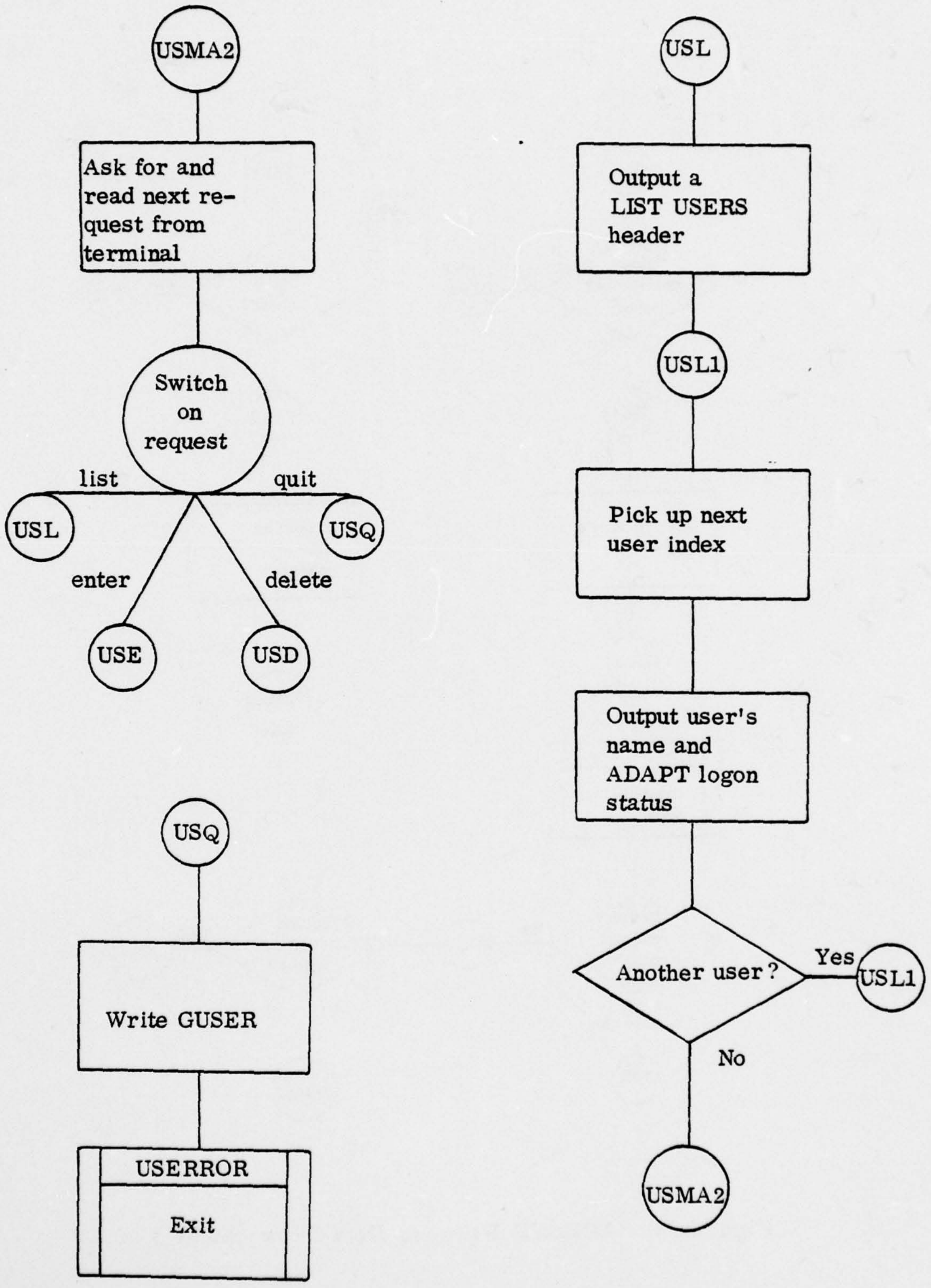


Figure 36. ADUSER Process Data Flow (Sheet 2 of 12)

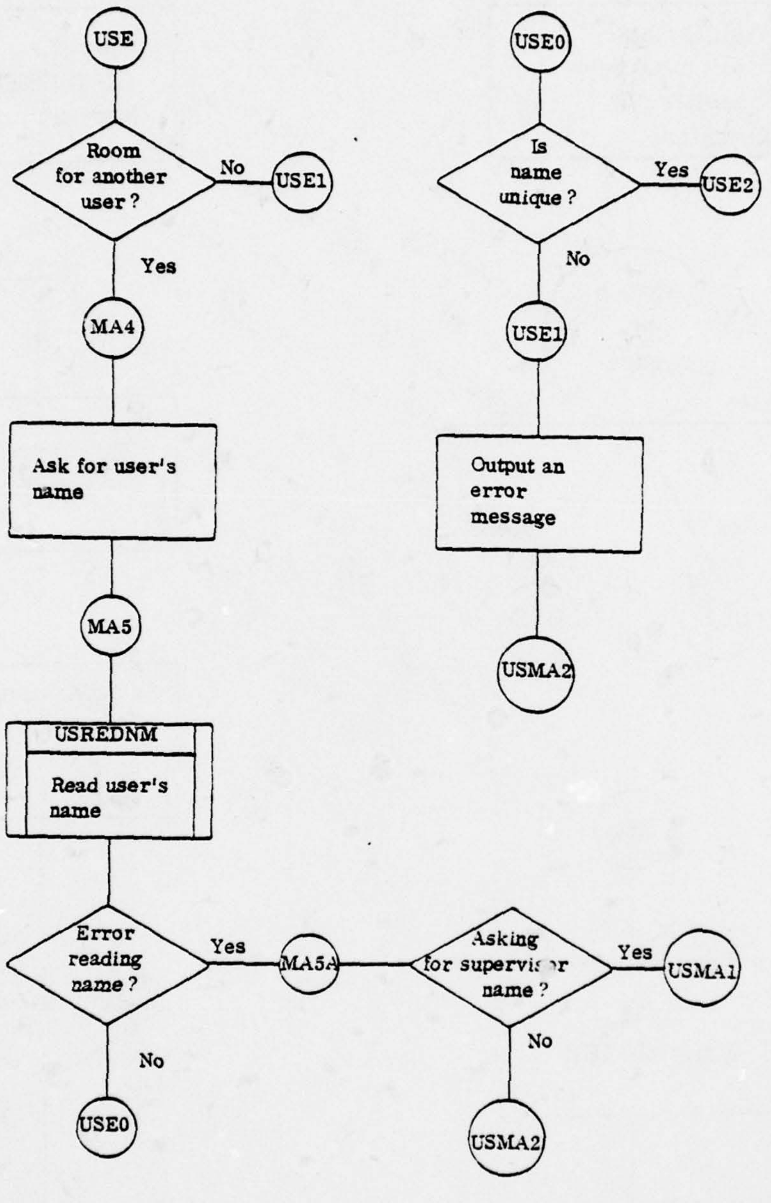


Figure 36. ADUSER Process Data Flow (Sheet 3 of 12)

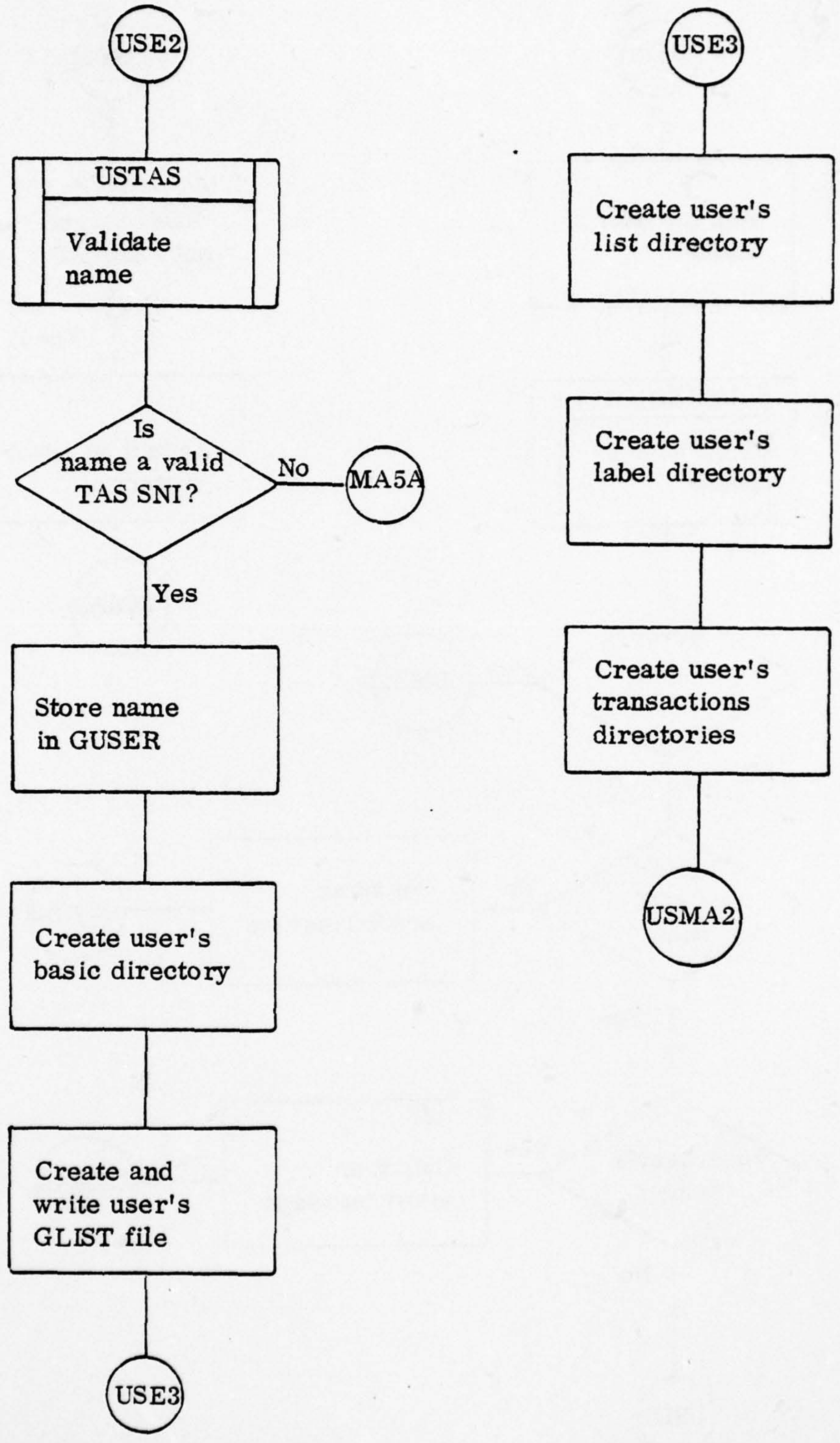

USE2
USTAS
Validate name
Is name a valid TAS SNI?
No
MA5A
Yes
Store name in GUSER
Create user's basic directory
Create and write user's GLIST file
USE3
USE3
Create user's list directory
Create user's label directory
Create user's transactions directories
USMA2

Figure 36. ADUSER Process Data Flow (Sheet 4 of 12)

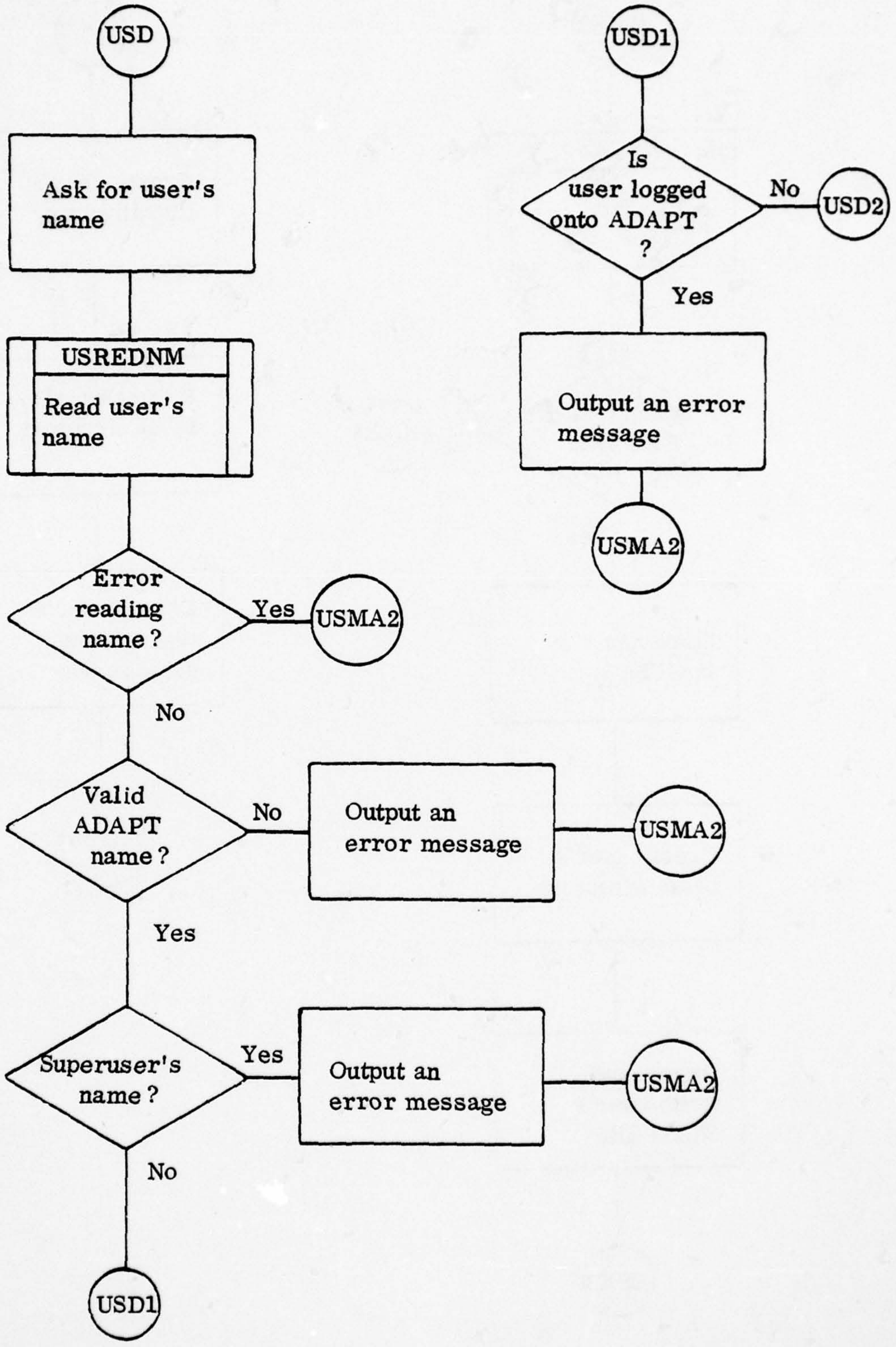


Figure 36. ADUSER Process Data Flow (Sheet 5 of 12)

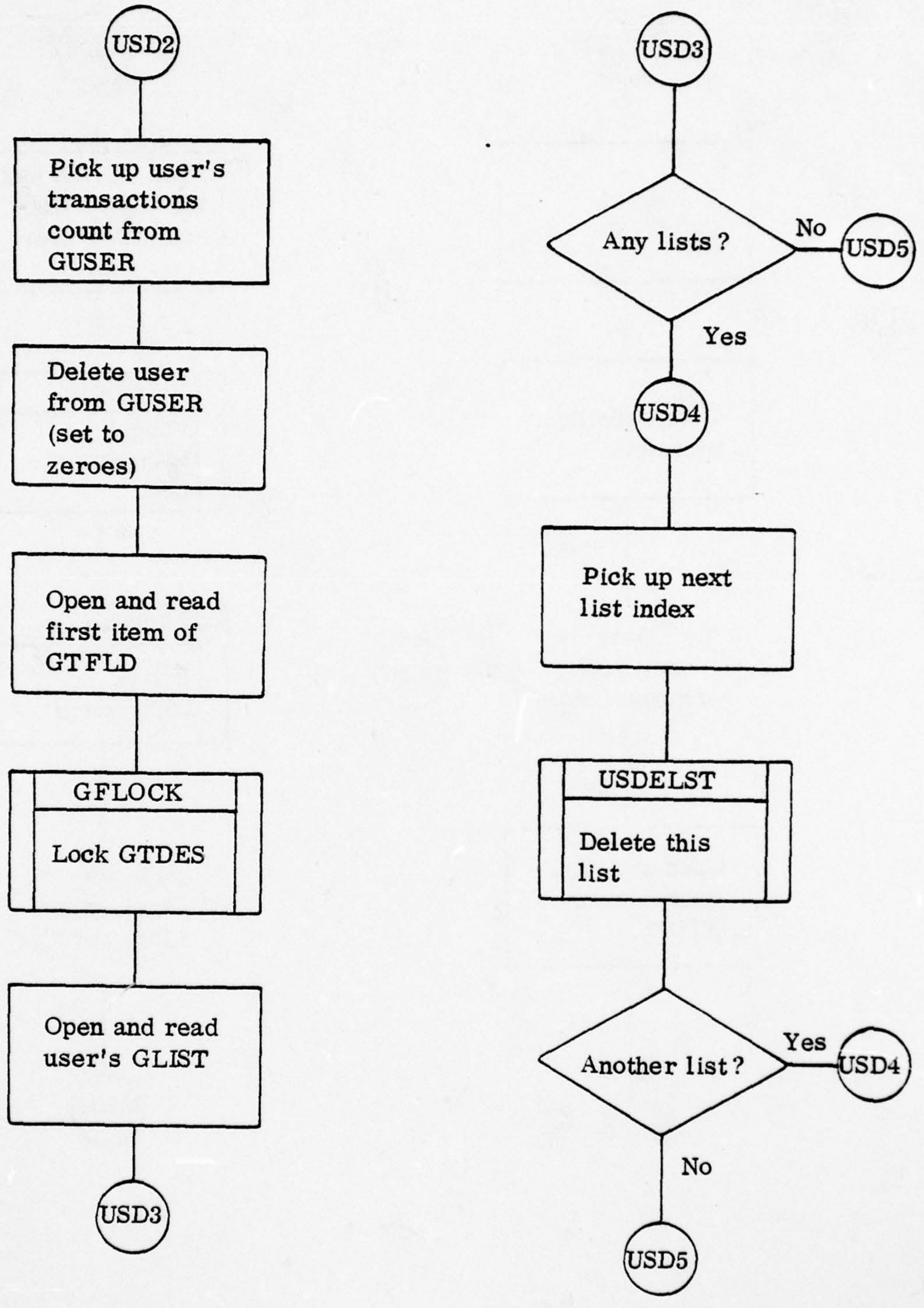

USD2
Pick up user's transactions count from GUSER
Delete user from GUSER (set to zeroes)
Open and read first item of GTFLD
GFLOCK
Lock GTDES
Open and read user's GLIST
USD3
USD3
Any lists ?
No
USD5
Yes
USD4
Pick up next list index
USDELST
Delete this list
Another list ?
Yes
USD4
No
USD5

Figure 36. ADUSER Process Data Flow (Sheet 6 of 12)

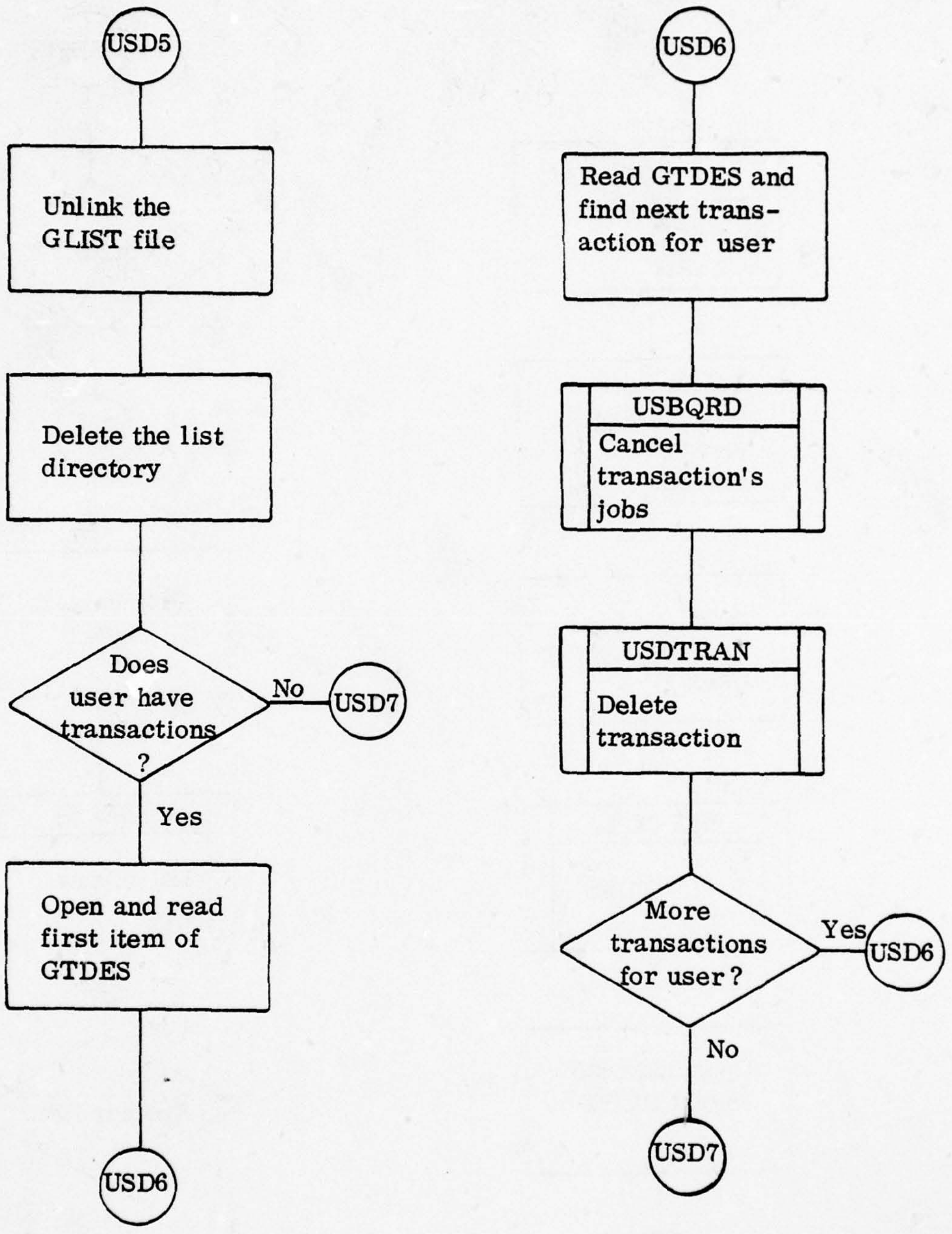

USD5
Unlink the GLIST file
Delete the list directory
Does user have transactions ?
No
USD7
Yes
Open and read first item of GTDES
USD6
USD6
Read GTDES and find next trans-action for user
USBQRD
Cancel transaction's jobs
USDTRAN
Delete transaction
More transactions for user ?
Yes
USD6
No
USD7

Figure 36. ADUSER Process Data Flow (Sheet 7 of 12)

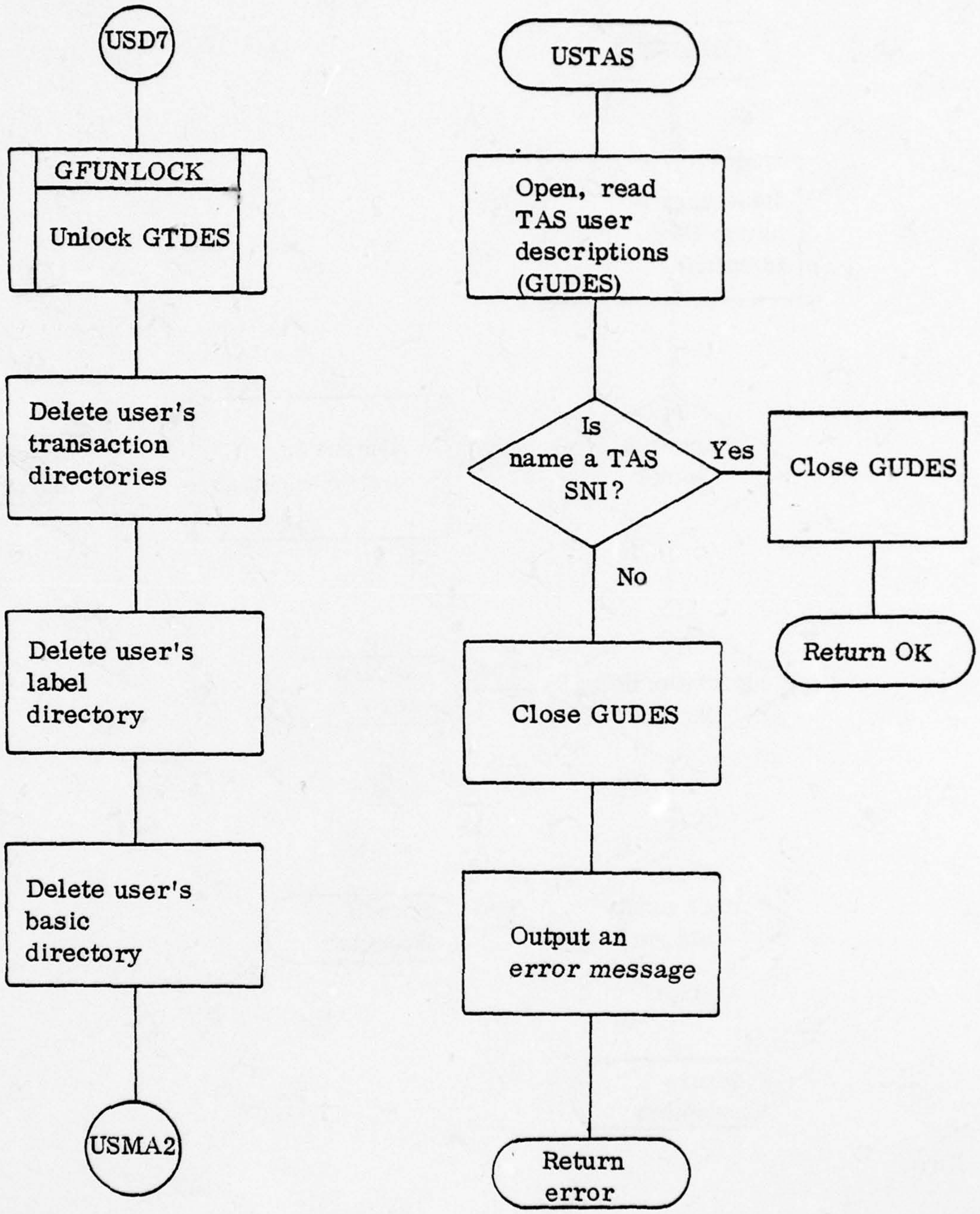

USD7
GFUNLOCK
Unlock GTDES
Delete user's transaction directories
Delete user's label directory
Delete user's basic directory
USMA2
USTAS
Open, read TAS user descriptions (GUDES)
Is name a TAS SNI?
Yes
Close GUDES
Return OK
No
Close GUDES
Output an error message
Return error

Figure 36. ADUSER Process Data Flow (Sheet 8 of 12)

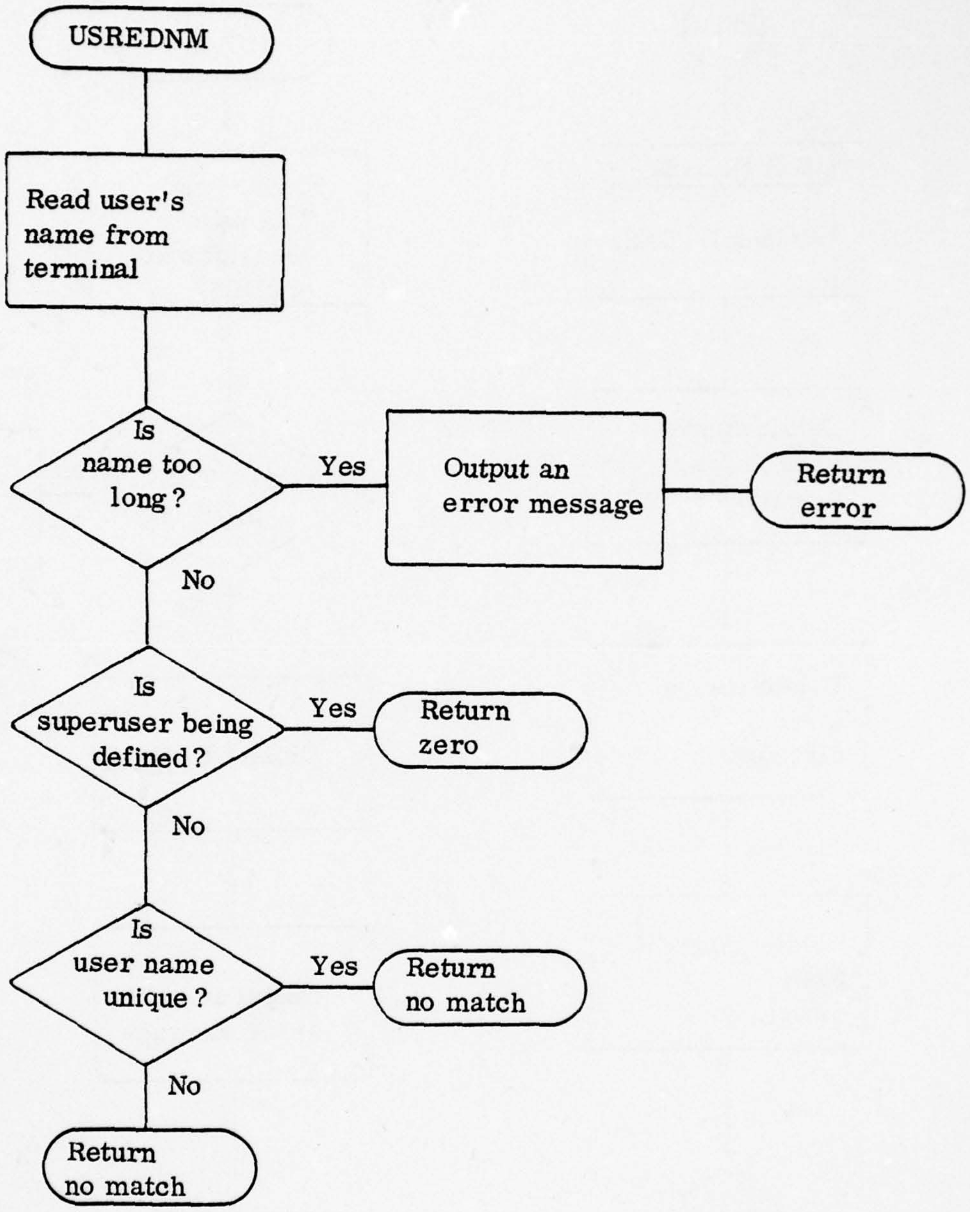
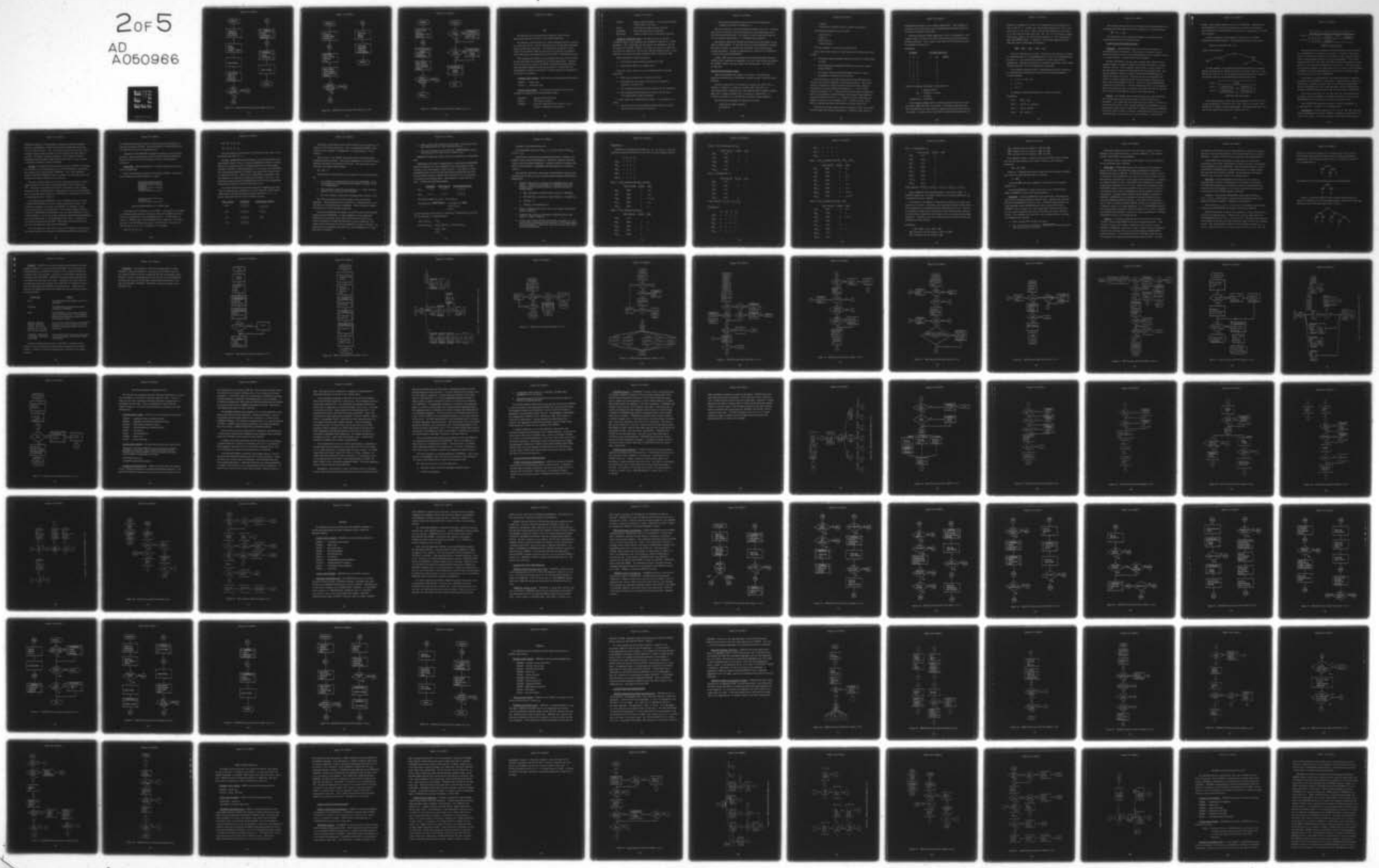


Figure 36. ADUSER Process Data Flow (Sheet 9 of 12)

AD-A050 966 LOGICON INC SAN DIEGO CALIF F/G 5/2
ADAPT I FINAL FUNCTIONAL AND SYSTEM DESIGN SPECIFICATION.(U)
JAN 78 L R ERICKSON, M E SOLEGLAD N00014-76-C-0899
UNCLASSIFIED 76-C-0899-2-F NL

2 OF 5
AD
A050966



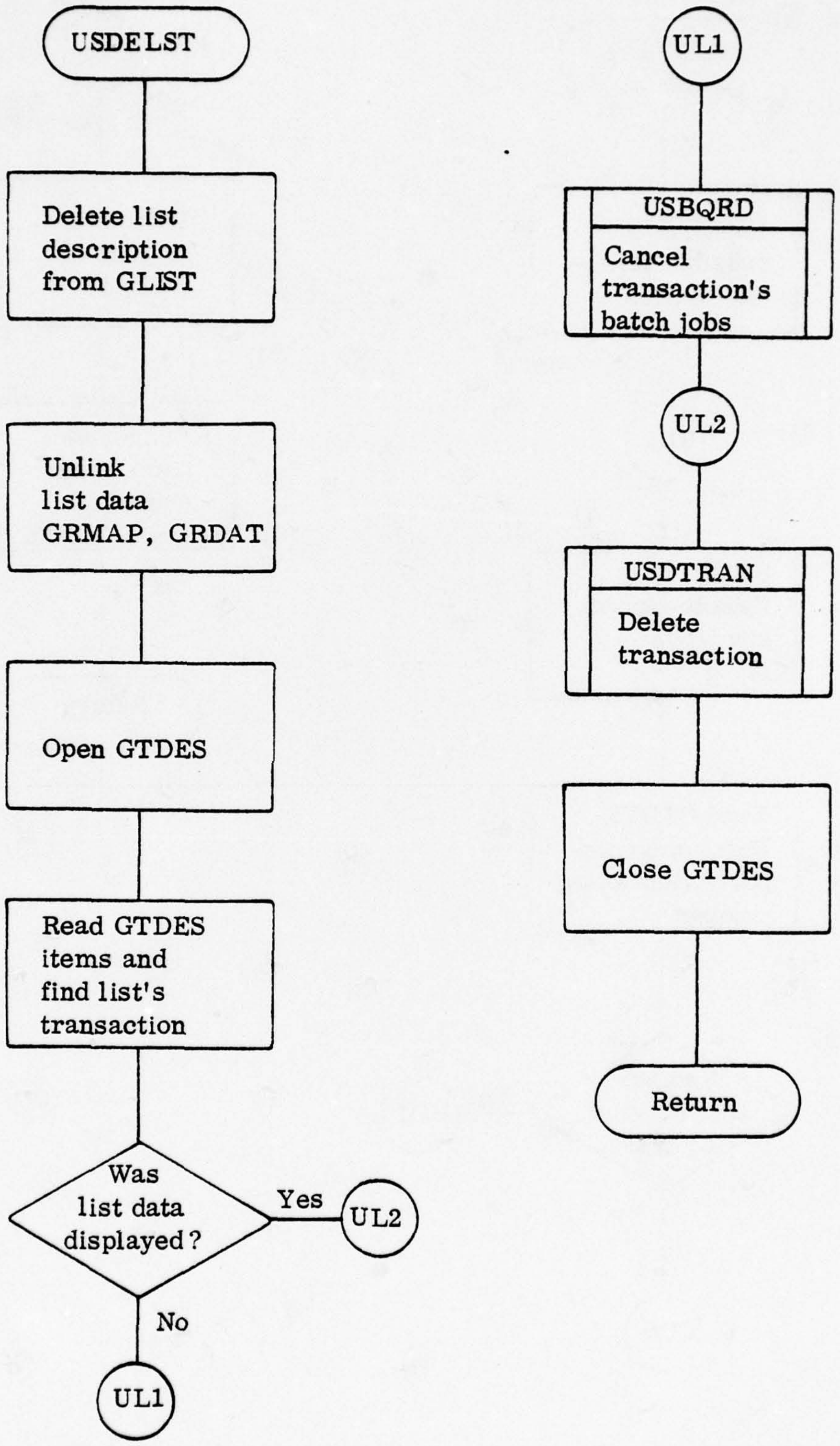

USDELST
Delete list description from GLIST
Unlink list data GRMAP, GRDAT
Open GTDES
Read GTDES items and find list's transaction
Was list data displayed?
Yes
UL2
No
UL1
UL1
USBQRD
Cancel transaction's batch jobs
UL2
USDTRAN
Delete transaction
Close GTDES
Return

Figure 36. ADUSER Process Data Flow (Sheet 10 of 12)

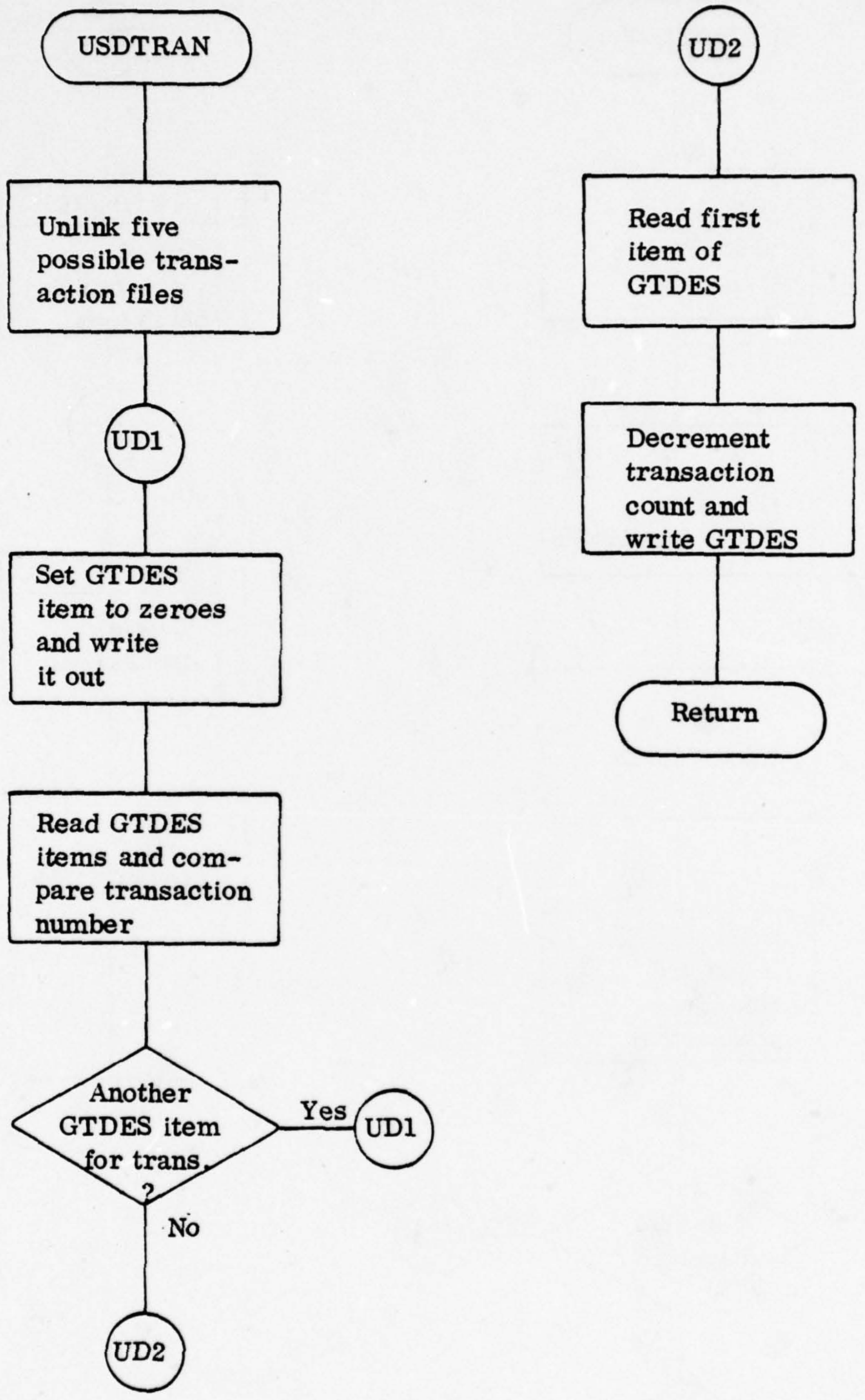

USDTRAN
Unlink five possible trans-action files
UD1
Set GTDES item to zeroes and write it out
Read GTDES items and com-pare transaction number
Another GTDES item for trans. ?
Yes
UD1
No
UD2
UD2
Read first item of GTDES
Decrement transaction count and write GTDES
Return

Figure 36. ADUSER Process Data Flow (Sheet 11 of 12)

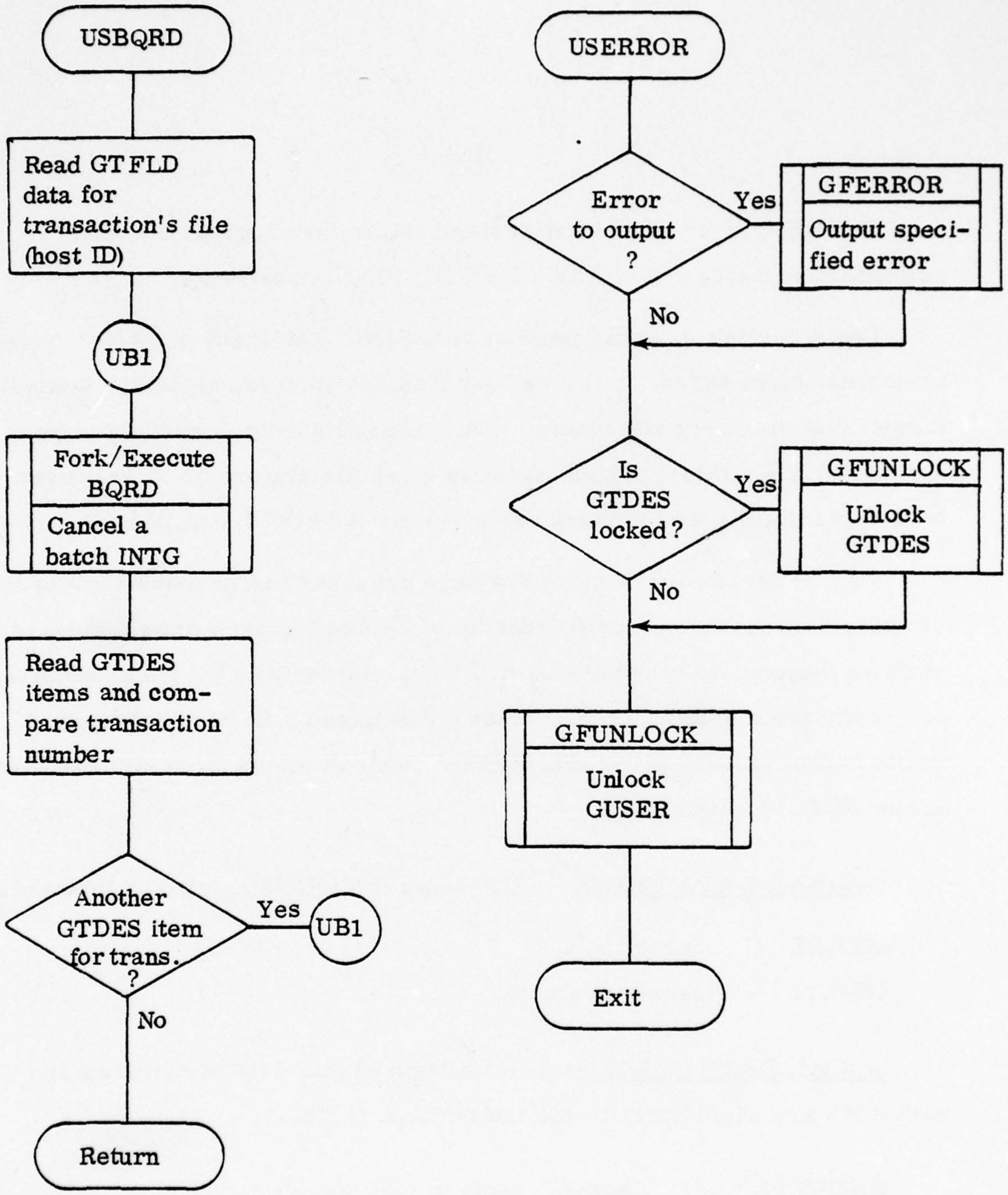


Figure 36. ADUSER Process Data Flow (Sheet 12 of 12)

## BEN

The BEN process normalizes and minimizes logical boolean expressions (selection criteria) in UDL FIND statements.

The selection criteria portion of a FIND statement must be converted to normal form before it can be transformed to a semantically equivalent target system query statement. This normalization operation is performed by the ADAPT I BEN process after the statement is validated both syntactically and semantically via the ADAPT I VALIDATE process.

This program description not only presents the process flow of BEN but also discusses the normalization of boolean expressions. Subjects such as disjunctive-normal-form (DNF), elementary boolean identities, and truth-table evaluation are briefly discussed. In particular, the implementation scheme is outlined for boolean expression normalization in the PDP-11 computers.

GLOBAL DATA USAGE – BEN uses the following global data arrays:

| | | |
|---|---|---|
| GTREE | – | parse tree. |
| GTDAT | – | parse tree data. |

LOCAL DATA USAGE – The following local data structures and variables are significant to the operations of BEN.

| | | |
|---|---|---|
| BNOPTBL | – | operator table structures. |
| BNXOR | – | XOR table structures. |
| BNTREE | – | parse tree data file which contains a copy of GTDAT to implement comparisons. |

| | | |
|---|---|---|
| BNVAR | – | unique variable pointers. Each scoped selection criteria has its own block. |
| BNTT | – | truth table for eight possible variables. |
| BNTTFIN | – | final truth table for the statement. |
| BNUMBYT | – | number of bytes need in truth table evaluation. |

<u>GENERAL PROCESS FLOW</u> – The BEN process is called by VALIDATE after it has validated the semantical contents of a FIND statement. The VALIDATE process passes ten arguments: the file descriptors for GTREE, GTDAT, four pipes and the user's output file, as well as the database ID, the user ID, and the validation mode. These arguments are stored and either utilized by BEN or passed to TSIG.

BEN performs two distinct functions:

a. Converting a general boolean expression to DNF.

b. Reducing the DNF to its minimal form.

Generally, these objectives are accomplished by the following operations:

a. BEN scans a parse tree for selection-criteria.

b. When detected, the boolean expressions are expanded to disjunctive-normal-form.

c. The resulting normalized boolean-expressions are minimized.

d. The minimized FIND statement is inserted into the parse tree replacing the original statement.

For this particular implementation scheme, two assumptions are made:

a. The parse tree representing the boolean expression has the proper operator precedence built into it.

b. The boolean expression cannot have more than eight unique variables (rel-terms) contained in it.

This last restriction is not particularly limiting to the user. A boolean expression or selection-criteria containing more than eight unique variables would be uncommon and, if necessary, could be expressed using two or more smaller expressions; i. e. , dependent interrogations.

Specifically, BEN reads GTREE and initializes the truth tables for the eight variables (BNTT). It calls the internal routine BNCENTR to identify the unique variables, to evaluate the truth table, and to minimize the boolean expression. Having accomplished this, the parse tree (BNTREE) is built by the function BNBLDPRS.

Finally, BEN executes the appropriate Target System Input Generator (TSIG) process, passing six arguments: the user ID and the file descriptors for GTREE, GTDAT, the user output file, and two pipes. See figure 37 for data flow.

## DISJUNCTIVE-NORMAL-FORM

Before the program description is discussed, a description of disjunctive-normal-form is presented to clarify the goal of the BEN major functions.

A boolean expression is in disjunctive-normal-form (DNF) when, in addition to variables, it contains no symbols other than those for conjunction (AND), disjunction (OR), and negation (NOT); negation symbols apply only to single variables; and no conjunct is a disjunction; i. e. , conjunction symbols occur only between variables or their negations.

The following are examples of expressions in DNF:

(A AND B) OR (NOT A AND B)

A OR NOT B

A

A AND B

A AND B OR A (based on UDL's operator precedence).

The following examples are not in DNF:

A XOR NOT B

A NOR A

NOT (A OR B)

(A OR B) AND A

The last example is in conjunctive-normal form.

There exist three reasons for reducing general boolean expressions to DNF:

a. Removing boolean operators that may not exist in certain target systems.

b. Removing boolean parentheses that may not exist in certain target systems.

c. Providing a standard (thus predictable) expression that is manageable in separate units (disjuncts).

The first reason is directed toward the XOR and NOR operators which are not directly mappable in most target systems. The second reason is important for those systems which allow only an implicit precedence for boolean expression evaluation; i. e., left-to-right, standard precedence, etc. The third reason has stronger ramifications since it is essential for transformations involving the scope-qualifier and mixed selection-criteria; i. e., selection-criteria containing geographic rel-terms, scope-qualifiers, etc. In some systems, these exist as separate statements and therefore must be factored out.

When general boolean expressions are reduced to DNF, they usually become larger (but simpler) expressions. Therefore, they should be minimized into smaller, more compact expressions. This compact expression can be used in place of the original general boolean expression since both are logically equivalent.

Reducing a general boolean expression to DNF is conceptually a very simple process. To derive the DNF of any general boolean expression, proceed as follows. First, make a truth-table for the boolean expression and variables contained in it.

EXAMPLE:

| Variables | | | Boolean Expression | | |
|---|---|---|---|---|---|
| A | B | C | A | $\oplus$ | $(\overline{B+CA})$ |
| 0 | 0 | 0 | | 1 | |
| 0 | 0 | 1 | | 1 | |
| 0 | 1 | 0 | | 0 | |
| 0 | 1 | 1 | | 0 | |
| 1 | 0 | 0 | | 0 | |
| 1 | 0 | 1 | | 1 | |
| 1 | 1 | 0 | | 1 | |
| 1 | 1 | 1 | | 1 | |

where the foregoing symbology is interpreted as:

| | | |
|---|---|---|
| - | = | negation (NOT). |
| $\oplus$ | = | exclusive-or (XOR). |
| + | = | or (OR). |
| juxtaposition | = | and (AND). |

The foregoing example shows a boolean expression with three variables where eight value sets are possible. The boolean expression has been evaluated and its truth-value bit-string is shown below the exclusive-or operator. In order to derive the DNF for this boolean expression, all disjuncts of variables A, B, and C are required where the expression is true (a value of 1). In this case, the boolean expression is true for five value sets of A, B, and C. That is, the boolean expression is true when A, B, and C have the values 000, 001, 101, 110, or 111 and only for these values (e.g., when A, B, and C have the values 0, 0, and 1 respectively, then $A \oplus (\overline{B+CA})$ equals 1). Therefore, the boolean expression can be expressed with the following five disjuncts:

$$\overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C + A\overline{B}C + AB\overline{C} + ABC$$

The above expression is in DNF. Note that in each disjunct a variable is complemented if its corresponding value is 0 and is not complemented if its value is 1. Also note that the exclusive-or (XOR) operator has disappeared. Although the expression is logically equivalent to the original boolean expression, it can be minimized considerably.

The technique utilized by BEN to minimize general DNF expressions is a modified Quine method which requires the following elementary boolean identities:

a. $A(B + C) = AB + AC$

b. $A + \overline{A} = 1$

c. $A1 = A$

An example using these identities is of value at this time.

EXAMPLE:

Step 1. $AB\overline{C} + ABC$

Step 2. $AB(\overline{C} + C)$, identity a

Step 3. $AB(1)$, identity b

Step 4. $AB$, identity c

With repeated (and selective) use of these three boolean identities, the foregoing large DNF expression can be minimized to the following form:

$$\overline{A}\overline{B} + AC + AB\overline{C}$$

The remainder of this program description is concerned with the implementation of the preceeding material.

## MAJOR FUNCTION DESCRIPTIONS

BNCENTL – The BNCENTL function acts as the controller for normalization and minimization. This function is called to evaluate either the selection criteria of the FIND statement or the selection criteria of a scoped operator, a subset of the FIND statement.

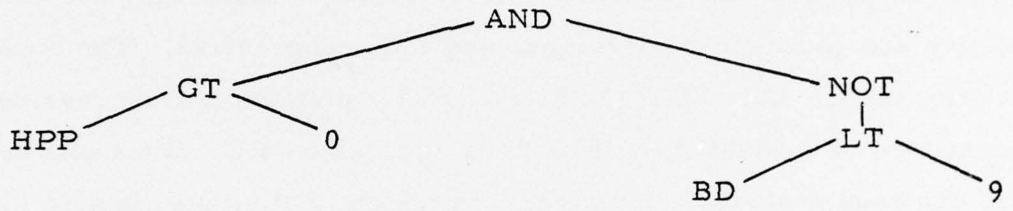
Initially, BNCENTL sets up a new variable table BNVAR which contains pointers to unique rel-terms of the FIND statement or a scoped selection criteria. The old operator table pointer is saved (if there is one) and a block of memory is allocated for the new operator table. The function BNOPR is called to build the operator table BNOPTBL and to identify the unique rel-terms. Once this has been accomplished, the number of unique variables which allow the truth table evaluation to be done by the routine BNTTE is known. Lastly, the function BNMINCEN minimizes the truth tables. With the minimization done, the old operator table is reestablished, the core memory associated with the new operator table is freed, and BNCENTL returns to the calling function.

BNOPR – The BNOPR function builds the operator table BNOPTBL. The core memory for the operator table is allocated and freed in BNCENTL. The components of the structure are as follows. BNOP is a boolean operator token. Both variables BNINDTY and BNOPIND are one-dimensional arrays with two values each. BNINDTY identifies BNOPIND's index type, either variable or internal. BNOPIND follows and is either a variable pointer, which refers to an index in BNVAR, or an internal pointer, which indexes another item entry in BNOPTBL. BNINDTY [0] and BNOPIND [0] reference the expression to the left of the operator, while BNINDTY [1] and BNOPIND [1] reference the expression to the right of the operator.

A visual schematic of the interplay of the parse tree (GTREE), BNOPTBL and BNVAR follows. The parsing of the boolean expression

HPP GT 0 AND NOT (BD LT 9)

yields the following tree:



For this example, the variable table BNVAR would have two entries: BNVAR [1] points to the GTREE token which represents HPP GT 0, and BNVAR [2] to BD LT 0. The following diagram represents the contents of the operator table for the aforementioned boolean expression:

| | | |
|---|---|---|
| Word 0 | BNOP | |
| Word 1 | BNINDTY [0] | BNINDTY [1] |
| Word 2 | BNOPIND [0] | BNOPIND [1] |

BNOPTBL Item Description

The boolean operator, AND, has a left expression of HPP GT 0 and a right expression of NOT (BD LT 9), which itself is the entry item BNOPTBL [1] (shown on the following page). The boolean operator, NOT, has but one expression which is the variable BD LT 9.

| | BNOP | BNINDTY [0] | BNOPIND [0] | BNINDTY [1] | BNOPIND [1] |
|---|---|---|---|---|---|
| BNOPTBL [0] | AND | Variable | Pointer to BNVAR [1] | Internal | Pointer to BNOPTBL [1] |
| BNOPTBL [1] | NOT | Variable | Pointer to BNVAR [2] | | |

BNOPTBL for Boolean Expression
HPP GT 0 AND NOT (BD LT 9)

The above discussion attempts to provide the purpose and the general flow of BNOPR but more specifics are needed. Four basic token types (stored in GTREE) are the basis for the operator table; they are boolean, relative and geographic operators, and scope aggregates. The boolean operators AND, OR, NOT, XOR, and NOR cause the left expression of the operator to be evaluated by BNOPR (a recursive call.) Once this expression has been evaluated, the right expression (if it exists) is also evaluated by BNOPR. Having accomplished this, the operator, the pointer-types, and the pointers are stored in BNOPTBL.

If the token is a scope qualifier, the scoped expression is analyzed and minimized by BNCENTL. At this point, scoped selection criteria are handled like a complete statement with its own BNVAR and BNOPTBL. Having accomplished this, the scope aggregate is placed in BNTREE by the function BNWRTV and the parse tree is further built by BNBLDPRS.

The relative operators (LT, GT, EQ, GE, LE, WRG and ORG) and geographic operators (INSIDE, OUTSIDE and ALONG) are processed by BNRELOPR before information is stored in the operator table.

After BNOPR has processed a boolean, relative, geographic, or scoped operator, it returns to the calling function.

<u>BNRELOPR</u> – Relative operators: LT, GT, LE, GE, EQ, WRG and ORG, and geographic operators: INSIDE, OUTSIDE, and ALONG are the bases of rel-terms. A rel-term consists of three parts: the relative/

geographic operator, the field name, and the data constant (character, numeric or geographic). Each rel-term must be identified and sorted from the other rel-terms. For rel-terms to be identical (and thus representing a single boolean variable), they must specify the same relative operator, field name, and data constant(s). Each unique rel-term is assigned a boolean variable index starting with 0 (up to 7), and a pointer to its token in GTREE is stored in BNVAR.

BNTTE – The BEN truth table evaluator function, BNTTE, determines the truth table of a scoped expression or a FIND statement by processing the contents of the operator table, BNOPTBL. One of the arguments passed to BNTTE is BNTPTR which is a pointer to a truth table where the results of the function BNTTE are to be stored.

Since up to eight variables are allowed, the truth-value set for these eight variables may be a set of bit-strings (BNTT) each 256 bits long ($2^8$). Therefore, each BNTT item must be 16 computer words in length. Boolean variable $V_0$ (BNVAR [0]) is represented by 16 words of alternating bits of zeros and ones, variable $V_1$ (BNVAR [1]) is represented by 16 words of alternating 00 and 11, and so on.

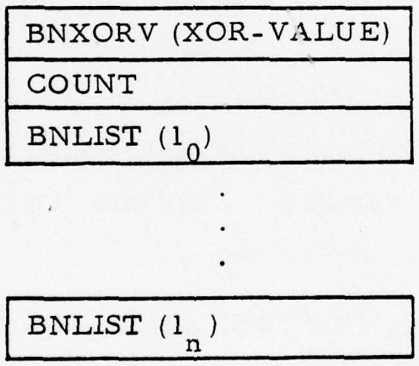
The operator table (BNOPTBL) is now evaluated using two 16-word variables (BNTTEMP[2] [16]) which contain intermediate results. The evaluation technique of an entry in BNOPTBL is dependent on the pointer types stored in the BNINDTYs. If the variable index k (pointer) is stored in BNOPTBL [x].BNOPIND[y], the truth table BNTT[k] is stored in BNTTEMP[y]. If on the other hand, the pointer BNOPTBL[x].BNOPIND[y] is the internal pointer j, the BNOPTBL[j] item entry is evaluated by recursively calling function BNTTE. The results of that evaluation are stored in the temporary truth table BNTTEMP[y].

Once the temporary truth table for the two BNTTEMPs is determined, the boolean operation as indicated by BNOPTBL[x]. BNOP is performed on two temporary truth tables with the result being stored in memory as indicated by pointer BNTPTR. Having evaluated the BNOPTBL[x] entry item BNTTE returns to the calling function.

Completion of the BNOPTBL evaluation provides a final truth table, BNTTFIN, which represents the truth-value of the boolean expression. This bit-string is used by the next phase of BEN, the normalizer/ minimizer.

BNMINCEN – The minimization of a boolean expression is done by function BNMINCEN.

Before discussing the operation of this phase of BEN, a description of the BNXOR table (shown below) is in order.



Pictorial Representation of a BNXOR Item.

This table has $2^n$ minterms for n variables. A minterm of n variables is a boolean product of these n variables, with each variable present in either its complemented or uncomplemented form. Conveniently, the index of a BNXOR item is the MINTERM value of that item. The right-most bit of the MINTERM represents $V_0$ (BNVAR [0]), the next bit $V_1$ (BNVAR [1]), and so on for n variables. For example:

0000 $\bar{V}_3\ \bar{V}_2\ \bar{V}_1\ \bar{V}_0$

. 0100 $\bar{V}_3\ V_2\ \bar{V}_1\ \bar{V}_0$

1011 $V_3\ \bar{V}_2\ V_1\ V_0$

The first minterm has the value 0, the second has the value 1, and the last has the value $(2^n - 1)$.

Initially, BNMINCEN allocates memory for the XOR table and sets each BNXORV field to $377_8$. In order to establish the DNF of a boolean expression, the truth-value bit-string generated by the Truth-Table-Evaluator (BNTTE) is analyzed bit by bit. For each bit set in BNTTFIN, the corresponding minterm's BNXORV (also called XOR-VALUE) field is cleared (set to zero). The first bit of BNTTFIN corresponds to the first minterm, the second bit to the second minterm, and so on. The minimization pass utilizes the BNXOR table directly once the applicable minterms have had their XOR-VALUEs set to zero.

The XOR-VALUE field is altered during the minimization process. When two minterms are simplified, one of the minterm's XOR-VALUE reflects that a variable has been effectively factored out. A "1" is placed in the XOR-VALUE in that variable's position.

| XOR-VALUE | MINTERM | Represented Disjunct |
|---|---|---|
| 0000 | $V_3V_2\bar{V}_1V_0$ | $V_3V_2\bar{V}_1V_0$ |
| 0100 | $V_3V_2\bar{V}_1V_0$ | $V_3\bar{V}_1V_0$ |
| 0101 | $V_3V_2\bar{V}_1V_0$ | $V_3\bar{V}_1$ |
| 1111 | $V_3V_2\bar{V}_1V_0$ | – |

Therefore, each minterm whose XOR-VALUE does not equal $377_8$ is a disjunct of the boolean expression. The corresponding MINTERM value represents the variables, complemented or uncomplemented, of this disjunct.

The structure of the BNXOR table lends itself to the minimization technique described earlier. The repeated application of the three boolean identities discussed is performed on the table. The three rules can be shortened to a single boolean identity.

$AB + A\bar{B} = A$

The mechanics of the utilization of the identity can be briefly outlined as:

a. Two disjuncts are found that have the same MINTERMs, where one and only one variable differs with respect to whether or not it is complemented.

b. This variable is factored out (by placing a "1" in that variable's position) in one of the two XOR-VALUEs.

c. The other disjunct is deleted by setting the XOR-VALUE to $377_8$.

The following text goes into greater depth on the minimization algorithm. Each minimization loop consists of a counting scheme, and a procedure for the minimization of minterms. These iterations continue until there are no more possible reductions.

The function BNCOUNT determines the COUNT value and the BNLIST indexes for each MINTERM. A count, tabulated for each applicable minterm $MT_x$ (a minterm whose XOR-VALUE does not equal $377_8$), denotes the number of minterms with which it can be simplified. The values of these minterms are stored in $BNLIST_x$ for future reference. To determine if two minterms ($MT_x$ and $MT_y$) can be simplified, two conditions must be satisfied:

a. Their current XOR-VALUEs must be equal. (This insures that the minterms have the same variables present.)

b. The value obtained from $(MT_x \oplus MT_y)(\overline{\text{XOR-VALUE}})$ contains one and only one bit which is set to one (any bit).

BNMAXCNT equals the largest count of any minterm for a particular pass.

Depending on the counts, simplification is performed by BNMINCEN. When two minterms are to be simplified, one of the minterm's XOR-VALUE is set to the resultant of XOR-VALUE + $(MT_x \oplus MT_y)$ which essentially factors out the variable that differs between $MT_x$ and $MT_y$. The other XOR-VALUE is deleted (XOR-VALUE is set to $377_8$) except in a special case discussed later. Both minterms' counts are set to zero. Consider the following example:

| | MINTERM | XOR-VALUE | Represented Disjunct |
|---|---|---|---|
| $MT_{15}$ | 1 1 1 1 | 1 0 0 0 | $V_2 V_1 V_0$ |
| $MT_5$ | 0 1 0 1 | 1 0 0 0 | $V_2 \bar{V}_1 V_0$ |

The XOR-VALUEs are equal. The result of

$$(MT_{15} \oplus MT_5)(\overline{\text{XOR-VALUE}}) = (1111 \oplus 0101)(\overline{1000})$$
$$= (1010)(0111)$$
$$= 0010$$

has one and only one bit set (bit 1); therefore, minterms $MT_{15}$ and $MT_5$ can be simplified, as indicated below.

$$\text{XOR-VALUE}_5 = 377_8$$
$$\text{XOR-VALUE}_{15} = (MT_{15} \oplus MT_5) + \text{XOR-VALUE}_{15}$$
$$= 1010 + 1000$$
$$= 1010$$

Variable $V_1$ has been factored out.

The represented disjunct of $MT_{15}$ (1 1 1 1) with an XOR-VALUE$_{15}$ = 1010 is $V_2\ V_0$.

The process of selecting which minterms should be combined is dependent on the counts of the minterms. Obviously, minterms with zero counts can not be further simplified. A minterm which can combine with only one other minterm is simplified with that minterm if it also has a count of 1.

All minterms which have counts equal to BNMAXCNT combine in the following way (let $MT_x$ be such a minterm thereby implying that COUNT$_x$ = BNMAXCNT):

a. Minimize $MT_x$ with those minterms in BNLIST$_x$ whose count equals 1. Under such conditions, the simplification process deviates from the normal procedure. The basic process is outlined below:

   1. $MT_z$ represents a minterm with a count of 1 in BNLIST$_x$.

   2. XOR-VALUE$_z$ is set equal to XOR-VALUE$_z$ + ($MT_x \oplus MT_z$).

   3. COUNT$_z$ = 0.

   4. COUNT$_x$ is decremented by 1.

b. Choose a minterm $MT_y$ whose count is the largest of those minterms in BNLIST$_x$.

c. Minimize $MT_y$ with those minterms in BNLIST$_y$ whose count equals 1. (Refer to step a.)

d. If after steps a and c COUNT$_y$ and COUNT$_x$ are equal to 1, $MT_y$ and $MT_x$ are not minimized with each other. (Their minimization would be redundant.) Otherwise, minimize $MT_x$ and $MT_y$ using the normal procedures.

EXAMPLE 1:

Assume four variables are present ($V_0$, $V_1$, $V_2$, and $V_3$). The five minterms whose XOR-VALUEs are not set to $377_8$ are minimized below.

| | $V_3$ | $V_2$ | $V_1$ | $V_0$ |
|---|---|---|---|---|
| $MT_0$ | 0 | 0 | 0 | 0 |
| $MT_1$ | 0 | 0 | 0 | 1 |
| $MT_8$ | 1 | 0 | 0 | 0 |
| $MT_9$ | 1 | 0 | 0 | 1 |
| $MT_{12}$ | 1 | 1 | 0 | 0 |

Pass 1 – $MT_8$ Combined with $MT_{12}$ and $MT_0$.

| | XOR-VALUE | COUNT | LIST |
|---|---|---|---|
| $MT_0$ | 0000 | 2 | 1, 8 |
| $MT_1$ | 0000 | 2 | 0, 9 |
| $MT_8$ | 0000 | 3 | 0, 9, 12 |
| $MT_9$ | 0000 | 2 | 1, 8 |
| $MT_{12}$ | 0000 | 1 | 8 |

Pass 2 – $MT_1$ Combined with $MT_9$.

| | XOR-VALUE | COUNT | LIST |
|---|---|---|---|
| $MT_0$ | 1111 | | |
| $MT_1$ | 0000 | 1 | 9 |
| $MT_8$ | 1000 | 0 | - |
| $MT_9$ | 0000 | 1 | 1 |
| $MT_{12}$ | 0100 | 0 | - |

Pass 3 – $MT_1$ Combined with $MT_8$.

| | XOR-VALUE | COUNT | LIST |
|---|---|---|---|
| $MT_0$ | 1111 | | |
| $MT_1$ | 1000 | 1 | 8 |
| $MT_8$ | 1000 | 1 | 1 |
| $MT_9$ | 1111 | | |
| $MT_{12}$ | 0100 | 0 | - |

Pass 4 – BNMAXCNT = 0.

| | XOR-VALUE | COUNT | LIST |
|---|---|---|---|
| $MT_0$ | 1111 | | |
| $MT_1$ | 1001 | 0 | - |
| $MT_8$ | 1111 | | |
| $MT_9$ | 1111 | | |
| $MT_{12}$ | 0100 | 0 | - |

Final reduction: $\overline{V}_2\ \overline{V}_1 + V_3\ \overline{V}_1\ \overline{V}_0$.

EXAMPLE 2:

| | $V_3$ | $V_2$ | $V_1$ | $V_0$ |
|---|---|---|---|---|
| $MT_0$ | 0 | 0 | 0 | 0 |
| $MT_2$ | 0 | 0 | 1 | 0 |
| $MT_4$ | 0 | 1 | 0 | 0 |
| $MT_7$ | 0 | 1 | 1 | 1 |
| $MT_{11}$ | 1 | 0 | 1 | 1 |

| | | | | |
|---|---|---|---|---|
| $MT_{12}$ | 1 | 1 | 0 | 0 |
| $MT_{13}$ | 1 | 1 | 0 | 1 |
| $MT_{15}$ | 1 | 1 | 1 | 1 |

Pass 1 – $MT_{15}$ Combined with $MT_7$, $MT_{11}$, $MT_{13}$

| | XOR-VALUE | COUNT | LIST |
|---|---|---|---|
| $MT_0$ | 0000 | 2 | 2, 4 |
| $MT_2$ | 0000 | 1 | 0 |
| $MT_4$ | 0000 | 2 | 0, 12 |
| $MT_7$ | 0000 | 1 | 15 |
| $MT_{11}$ | 0000 | 1 | 15 |
| $MT_{12}$ | 0000 | 2 | 4, 13 |
| $MT_{13}$ | 0000 | 2 | 12, 15 |
| $MT_{15}$ | 0000 | 3 | 7, 11, 13 |

Pass 2 – $MT_0$ Combined with $MT_2$, $MT_4$.

| | XOR-VALUE | COUNT | LIST |
|---|---|---|---|
| $MT_0$ | 0000 | 2 | 2, 4 |
| $MT_2$ | 0000 | 1 | 0 |
| $MT_4$ | 0000 | 2 | 0, 12 |
| $MT_7$ | 1000 | 0 | - |
| $MT_{11}$ | 0100 | 0 | - |
| $MT_{12}$ | 0000 | 1 | 4 |
| $MT_{13}$ | 1111 | | |
| $MT_{15}$ | 0010 | 0 | - |

Pass 3 – BNMAXCNT = 0.

| | XOR-VALUE | COUNT | LIST |
|---|---|---|---|
| $MT_0$ | 1111 | 0 | - |
| $MT_2$ | 0010 | 0 | - |
| $MT_4$ | 1111 | | |
| $MT_7$ | 1000 | 0 | - |
| $MT_{11}$ | 0100 | 0 | - |
| $MT_{12}$ | 1000 | 0 | - |
| $MT_{13}$ | 1111 | | |
| $MT_{15}$ | 0010 | 0 | - |

Final reduction: $\overline{V}_3\overline{V}_2\overline{V}_0 + V_2V_1V_0 + V_3V_1V_0 + V_2\overline{V}_1\overline{V}_0 + V_3V_2V_0$.

Upon completion of the foregoing steps, a new count is made. If MAXCNT equals zero, no further reductions can be made; otherwise, the minimization process continues.

The reason for calculating a simplification count prior to actual minimization is as follows. Intuitively, one would think it doesn't really matter which minterms are combined during the minimization process. However, it is possible to make initial combinations which inhibit multiple combinations that may be possible in later passes, thus not producing the minimal expression. Therefore, only those minterms which combine with the most minterms are used initially.

EXAMPLE:

$AB\overline{C} + \overline{A}\overline{B}C + ABC + A\overline{B}\overline{C} + A\overline{B}C$

$AB\overline{C}$ combines with two disjuncts, $ABC$ and $A\overline{B}\overline{C}$.

$\overline{A}\overline{B}C$ combines with one disjunct, $A\overline{B}C$.

$ABC$ combines with two disjuncts, $AB\overline{C}$ and $A\overline{B}C$.

$A\overline{B}\overline{C}$ combines with two disjuncts, $AB\overline{C}$ and $A\overline{B}C$.

$A\overline{B}C$ combines with two disjuncts, $\overline{A}\overline{B}C$ and $A\overline{B}\overline{C}$.

If an attempt is made to simplify using disjunct $\overline{A}\overline{B}C$ (which combines with only one other disjunct), the result is reduced to the following expression:

$$\overline{B}C + A\overline{C} + ABC$$

However, if simplification uses the disjuncts with the largest combination counts, the following expression is produced:

$$A + \overline{A}\overline{B}C$$

Notice that $\overline{A}\overline{B}C$ was never simplified, but the other four disjuncts reduced to A.

Once the BNMAXCNT is equal to zero (after a new count has been made), BNMINCEN returns to the calling function.

<u>BNCOUNT</u> – Function BNMINCEN calls BNCOUNT to determine and record which minterms simplify with each other. Initially, the minterms' COUNTs and BNMAXCNT are set to zero. BNCOUNT next performs the count algorithm on each BNXOR[X] item ($MT_x$) whose XOR-VALUE (BNXORV) does not equal $377_8$. The values of the minterms with which the $MT_x$ can be simplified are stored in $BNLIST_x$ and a tally, $COUNT_x$, is kept. Two minterms ($MT_x$ and $MT_y$) can be simplified if the following two conditions are satisfied.

a. Their current XOR-VALUEs are equal.

b. The resultant from ($MT_x \oplus MT_y$) ($\overline{\text{XOR-VALUE}}$) contains one and only one bit which is set to one.

When these conditions are met, the index value of $MT_x$ is stored in $BNLIST_y$ and the index value $MT_y$ is stored in $BNLIST_x$. Both counters $COUNT_x$ and $COUNT_y$ are incremented.

BNCOUNT returns a value which is the largest count of any minterm for this particular pass of BNXOR.

<u>BNBLDPRS</u> – BNBLDPRS directs the rebuilding of the statement's parse tree into the array BNTREE using the appropriate minterms in BNXOR and pointers in BNVAR. The BNXOR structure contains a series of minterms, whose BNXORV are not set to $377_8$, that were used to form the disjunctive-normal-form of the original boolean expression. Having set the minterm counter to zero, BNBLDPRS cycles through each minterm, locating those minterms not set to $377_8$. When such a minterm is found, BNBLDPRS ensures (via function BNWRTV) that each set rel-term has a representative set of nodes in BNTREE and that the minterm counter is incremented. Next the function BNDJCT places BNXOR's disjuncts in BNTREE. ($DISJUNCT_x$ is the resultant of logically ORing the two bit strings $MINTERM_x$ and $XOR\text{-}VALUE_x$.) Once BNDJCT has returned, a test is made to determine whether BNXOR represents the minimization of a selection criteria of a scoped aggregate. If it is, BNBLDPRS returns to the calling function; otherwise, the non-terminal node for FIND is inserted into BNTREE and BNTREE is written out, using the pathname for GTREE. BNBLDPRS returns to the calling function.

<u>BNDJCT</u> – The function BNDJCT places the disjuncts of the XOR-VALUEs, stored in the BNXOR structure, into parse tree form. If the selection criteria is not scoped, a minterm's XOR-VALUE is put in BNTREE by BNMTRM. After this is done, a check is made to determine whether other minterms have to be processed (based on BNBLDPRS's minterm counter). If not, BNDJCT returns a BNTREE pointer for the non-terminal node representing the minterm just processed. If at least

one minterm still needs to be processed, BNDJCT is called recursively, and once it has returned a pointer, an OR non-terminal node is placed in BNTREE. A pointer to this OR node is returned to the calling function.

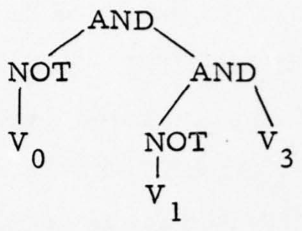
If the BNXOR table represents a scoped selection criteria, the disjunctive-normal-form of the selection criteria is placed in the operator table BNOPTBL instead of BNTREE. Each individual minterm is stored in BNTREE by BNMTRM and is treated henceforth as a single, unique rel-term with its own entry in BNVAR. In this situation, the function BNDJCT does not return a value to the calling function.

BNMTRM – Function BNMTRM builds the parse tree nodes for minterms which consists of ANDing the set rel-terms of the minterm. The BNTREE presentation of a minterm consists of AND and NOT non-terminal nodes whose pointers to rel-terms (variables) are retrievable from the rel-term's BNVBEN pointer in BNVAR.

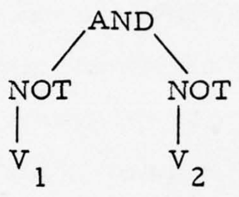
Example 1 of the function BNMINCEN, whose final reduction is $V_1V_2 + V_0V_1V_3$, yields itself to a discussion and demonstration of the purpose of functions BNDJCT, BNMTRM, and BMWRTV.

Each rel-term $V_1$, $V_2$, $V_3$, and $V_0$ has a representative set of nodes in the parse tree BNTREE and a pointer to that node in BNVAR.

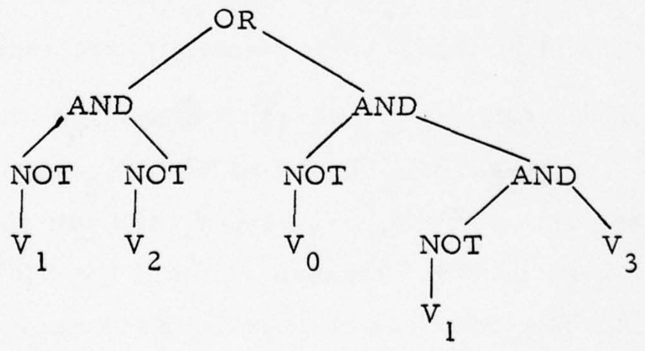
BNMTRM builds the two parse tree segments for $\overline{V}_1\overline{V}_2$ (minterm $MT_1$) and $\overline{V}_0\overline{V}_1V_3$ (minterm $MT_{12}$). Minterm $MT_{12}$ has the XOR-VALUE of 0100 meaning rel-terms $V_3$, $V_1$, and $V_0$ are set. The logical OR operation is performed on the $MT_{12}$ bit map 1100 and the XOR-VALUE 0100 resulting in 1100. The only positions of interest in the resultant are again $V_3$, $V_1$, and $V_0$, which for $V_0$ and $V_1$ is a zero value and for $V_3$ is one. When the value for a rel-term's position is the resultant zero, the rel-term is NOTed before being ANDed with other set rel-terms in the minterm. Algebraically, the minimized minterm $MT_{12}$ is $V_3\overline{V}_1\overline{V}_0$ and its parse tree representation is:



The minterm $MT_1$ has the following parse tree representation:



Finally, it is function BNDJCT which actually connects these two parse tree segments via an OR non-terminal node, and hence the following final parse tree for the expression $\overline{V}_1\overline{V}_2 + \overline{V}_0\overline{V}_1V_3$.



<u>BNWRTV</u> – BNWRTV writes non-terminal and terminal nodes representing variables (rel-terms) stored in BNTREE. The calling function BNBLDPRS passes to BNWRTV the pointer for the original GTREE non-terminal node for the variable. Each leg of a non-terminal token points to another node of BNTREE. BNWRTV investigates these nodes and acts according to the token type of the node. Frequently, the legs of the non-terminal node cause the recursive call to BNWRTV to establish the necessary new leg pointers in BNTREE as indicated below. These new leg pointers are temporarily stored in BNLIST until the current node is written into BNTREE.

| Token Type | Action |
|---|---|
| POLY | Call BNWGEOL with a pointer to POLY's GEOLIST. |
| GEOLIST | Call BNWGEOL and return the pointer returned by BNWGEOL. |
| FIND | Call BNWRTV to write the non-terminal node for SOURCE or IN. Its second leg points to the disjunctive-normal-form of the selection criteria. |
| CIRCLE, ROUTE, INSIDE, OUTSIDE, ALONG, WRG, ORG, SNAME, LT, LE, GT, GE, EQ, IN, SOURCE | For each leg of these tokens, call BNWRTV and store its result in the array BNLIST. BNWRTV returns the pointer to this terminal node. |
| INTEGER, CHARCONS, NUMCONS, AGGNAME, FLDNAME | A terminal node for any one of these token types is essentially copied from GTREE to BNTREE. |

The non-terminal nodes are built in BNTREE, storing the token, number of legs, and the pointers which were temporarily kept in array BNLIST. A pointer to the non-terminal node is returned to the calling function.

BNWGEOL – The writing out of the non-terminal node for token GEOLIST is done by BNWGEOL. Each leg of the GEOLIST results in the calling of BNWRTV and the storing into BNLIST of the pointer which BNWRTV returns. Having processed each leg, the actual GEOLIST non-terminal node is placed in BNTREE, transferring the pointers for the legs from BNLIST to BNTREE. BNWGEOL returns the pointer to the GEOLIST node.

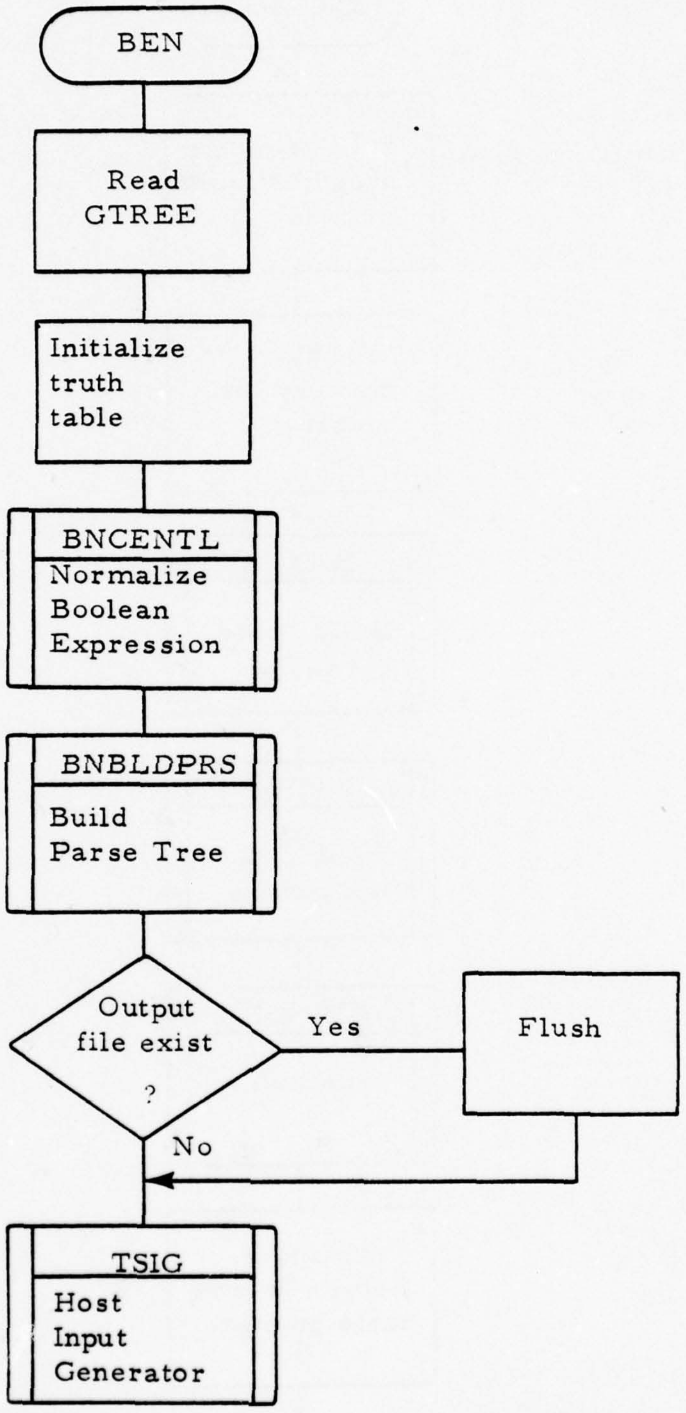


Figure 37. BEN Process Data Flow (Sheet 1 of 13)

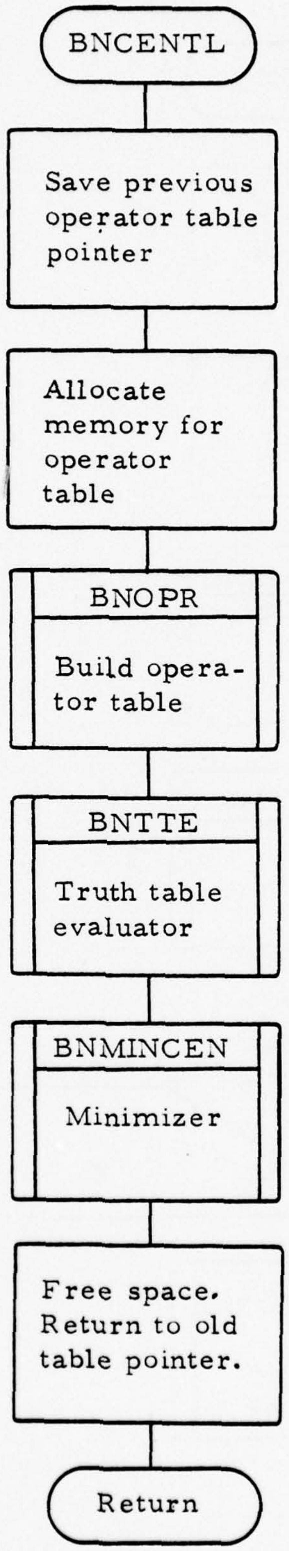

BNCENTL
Save previous operator table pointer
Allocate memory for operator table
BNOPR
Build operator table
BNTTE
Truth table evaluator
BNMINCEN
Minimizer
Free space. Return to old table pointer.
Return

Figure 37. BEN Process Data Flow (Sheet 2 of 13)

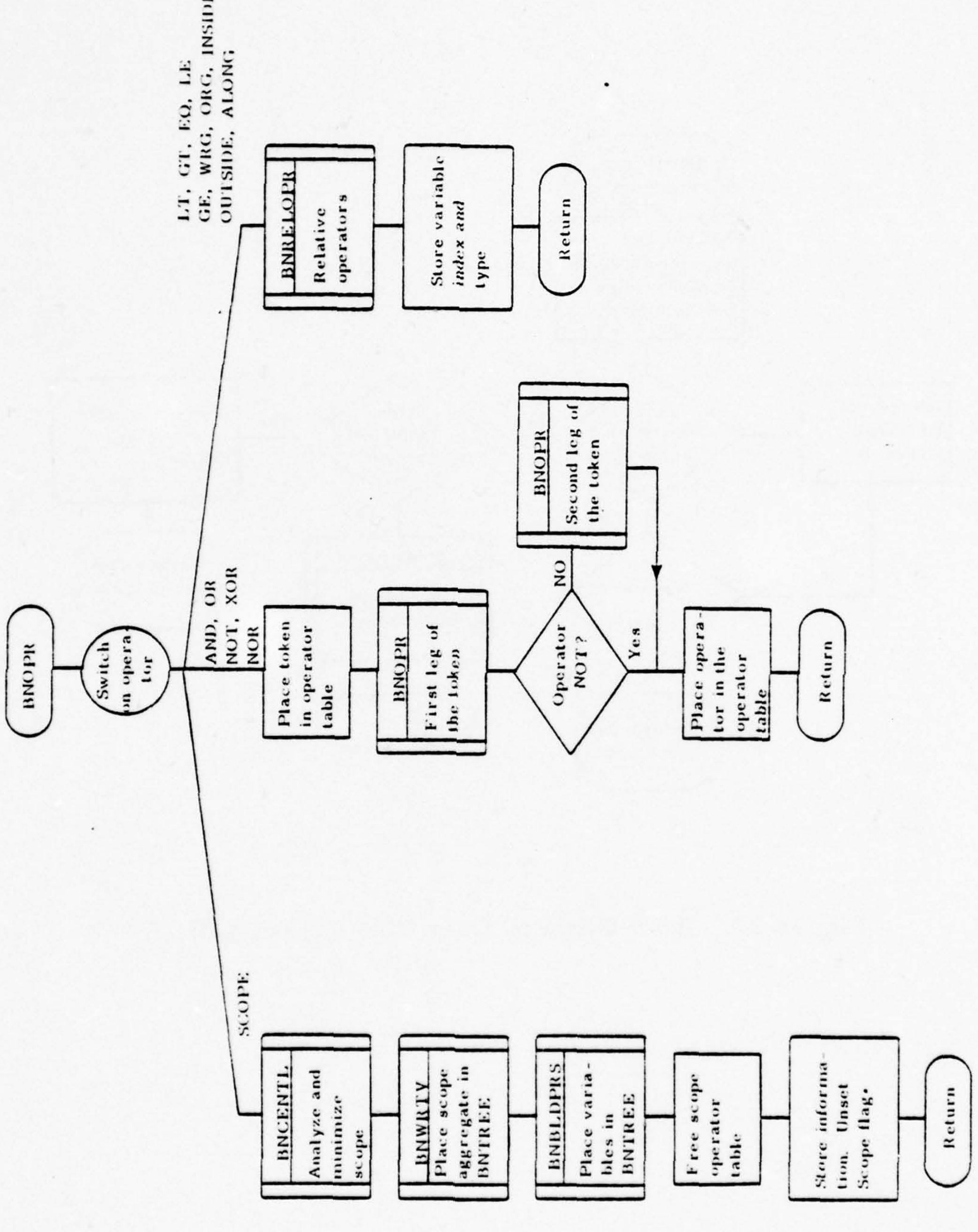

BNOPR
Switch on opera- tor
SCOPE
BNCENTL
Analyze and minimize scope
BNWRTY
Place scope aggregate in BNTREE
BNBLDPRS
Place varia- bles in BNTREE
Free scope operator table
Store informa- tion. Unset Scope flag.
Return
AND, OR NOT, XOR NOR
Place token in operator table
BNOPR
First leg of the token
Operator NOT?
NO
BNOPR
Second leg of the token
Yes
Place opera- tor in the operator table
Return
LT, GT, EQ, LE GE, WRG, ORG, INSIDE OUTSIDE, ALONG
BNRELOPR
Relative operators
Store variable index and type
Return

Figure 37. BEN Process Data Flow (Sheet 3 of 13)

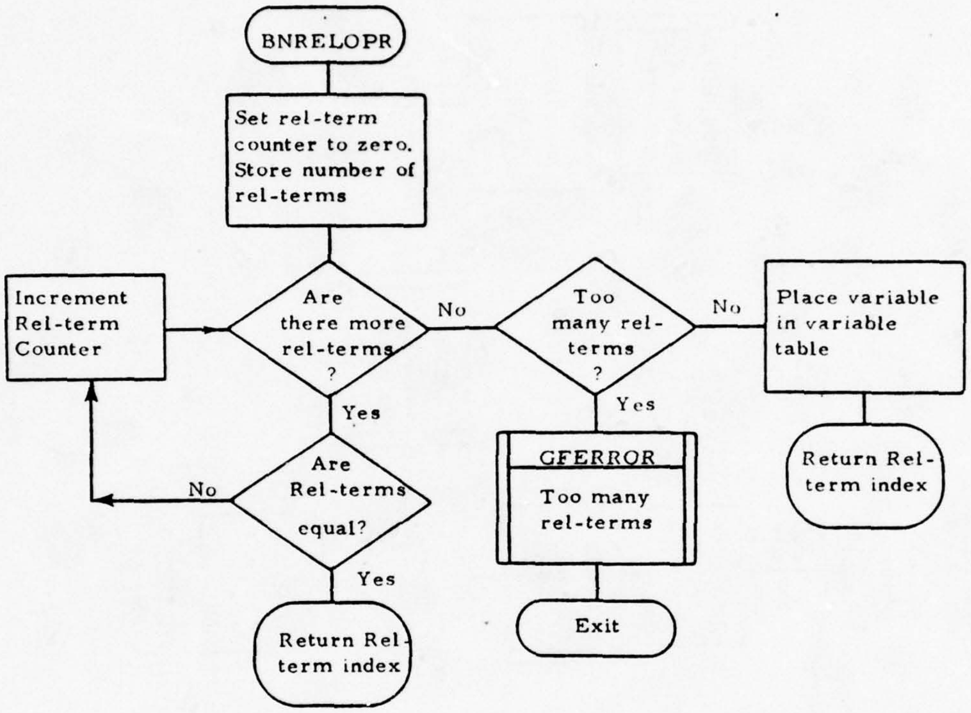

BNRELOPR
Set rel-term counter to zero. Store number of rel-terms
Increment Rel-term Counter
Are there more rel-terms ?
No
Yes
Are Rel-terms equal?
No
Yes
Return Rel term index
Too many rel-terms ?
No
Yes
GFERROR
Too many rel-terms
Exit
Place variable in variable table
Return Rel-term index

Figure 37. BEN Process Data Flow (Sheet 4 of 13)

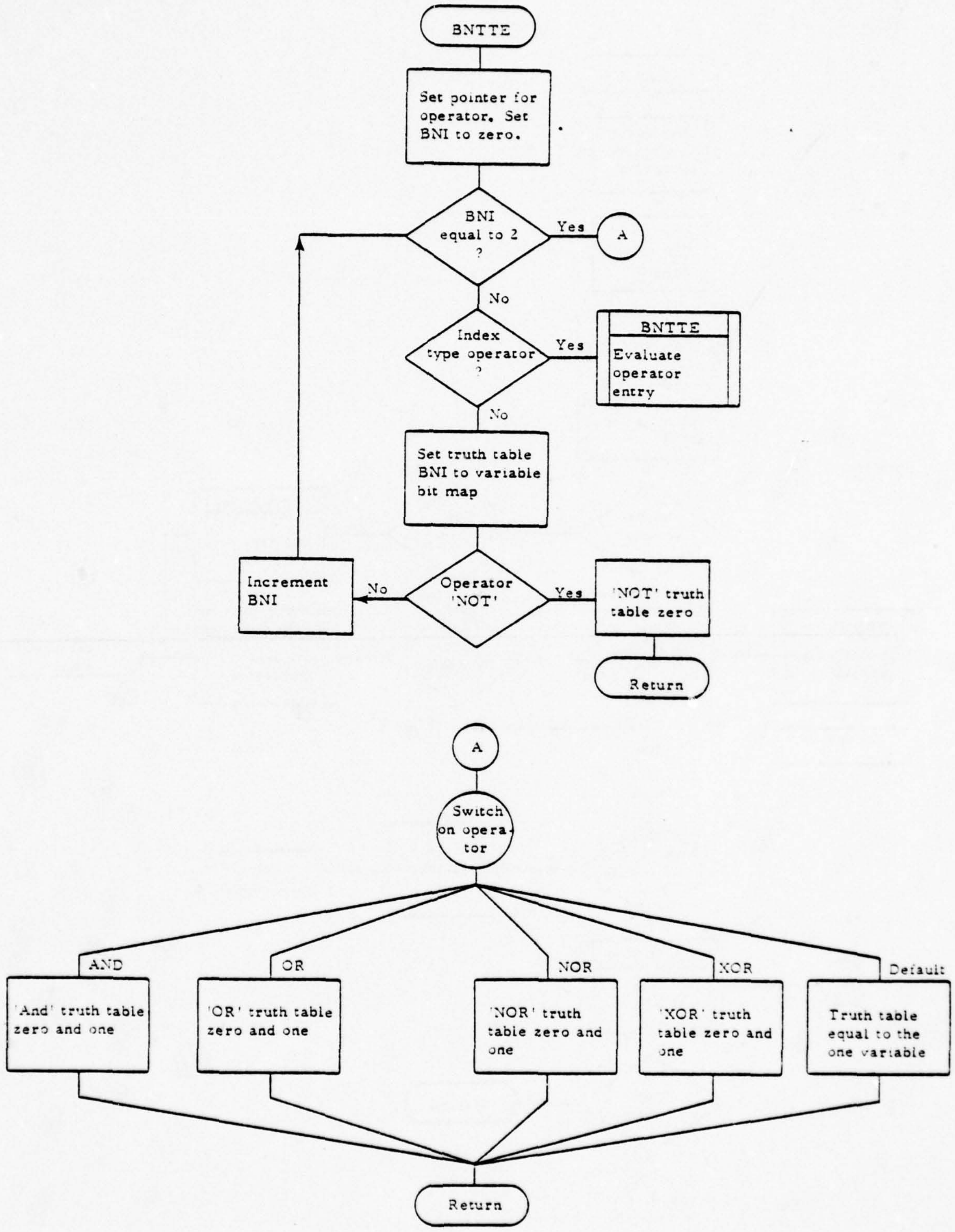

BNTTE
Set pointer for operator. Set BNI to zero.
BNI equal to 2 ?
Yes
A
No
Index type operator ?
Yes
BNTTE
Evaluate operator entry
No
Set truth table BNI to variable bit map
Increment BNI
No
Operator 'NOT'
Yes
'NOT' truth table zero
Return
A
Switch on operator
AND
OR
NOR
XOR
Default
'And' truth table zero and one
'OR' truth table zero and one
'NOR' truth table zero and one
'XOR' truth table zero and one
Truth table equal to the one variable
Return

Figure 37. BEN Process Data Flow (Sheet 5 of 13)

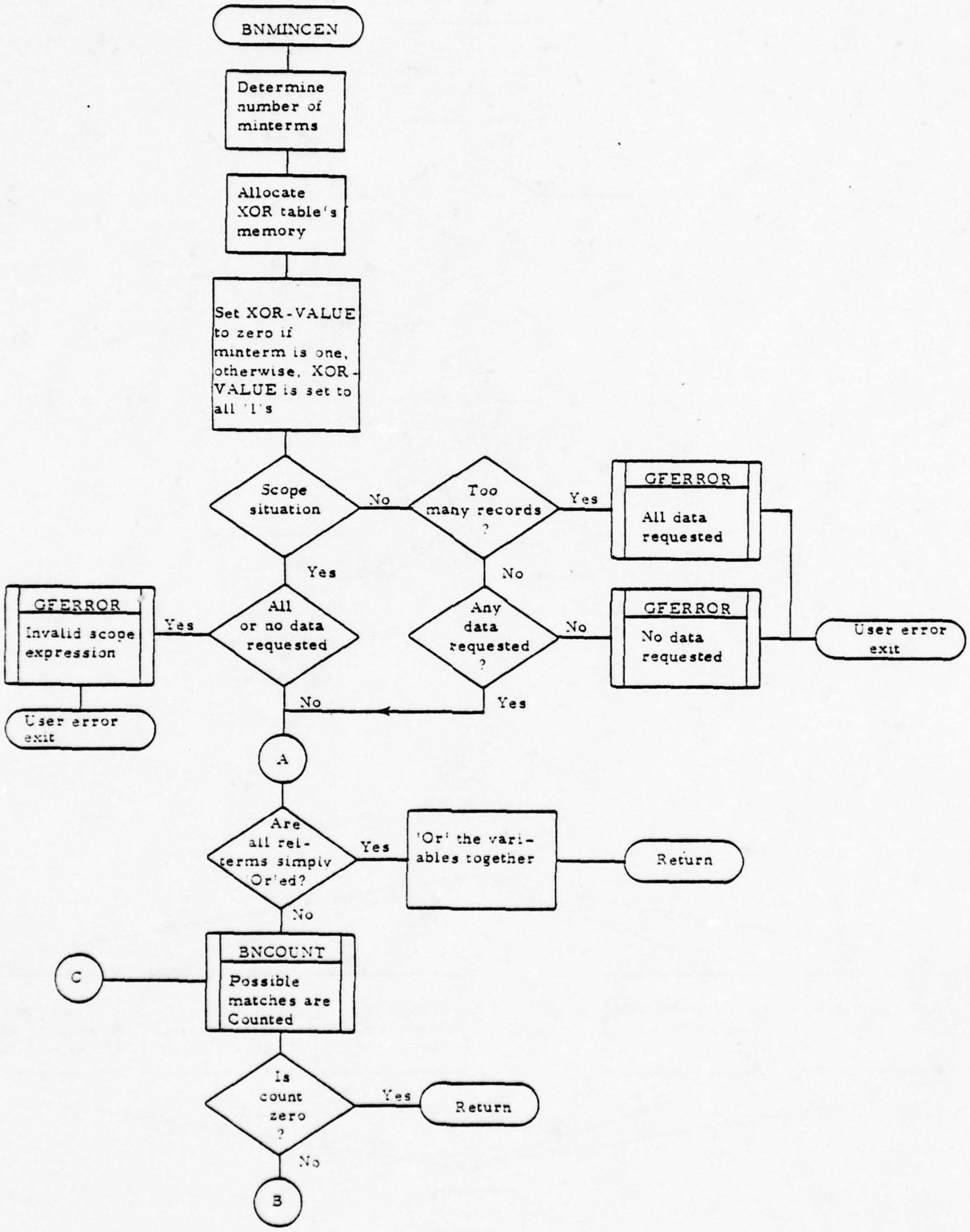


Figure 37. BEN Process Data Flow (Sheet 6 of 13)

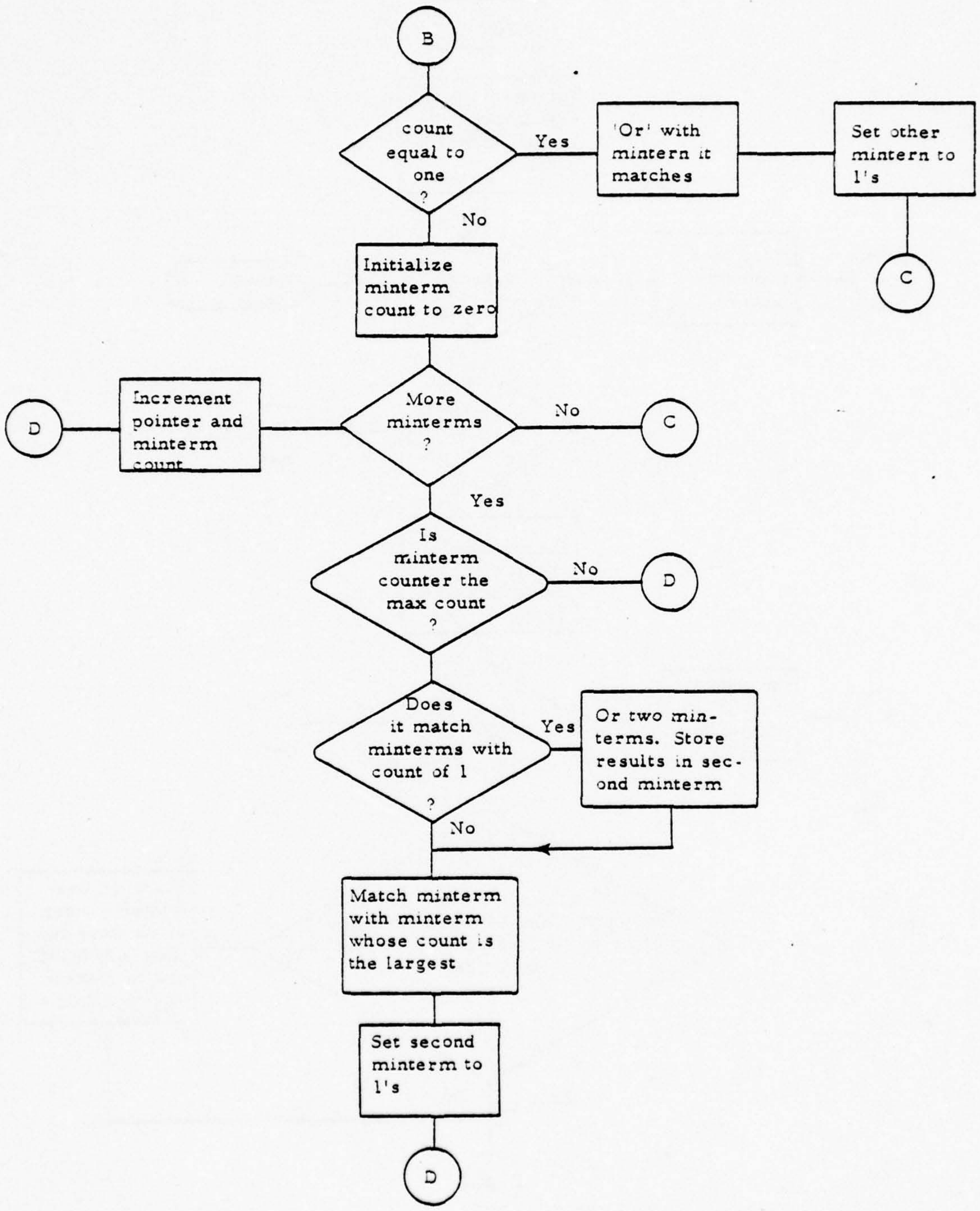

B
count equal to one ?
Yes
'Or' with mintern it matches
Set other mintern to 1's
C
No
Initialize minterm count to zero
Increment pointer and minterm count
D
More minterms ?
No
C
Yes
Is minterm counter the max count ?
No
D
Does it match minterms with count of 1 ?
Yes
Or two min-terms. Store results in sec-ond minterm
No
Match minterm with minterm whose count is the largest
Set second minterm to 1's
D

Figure 37. BEN Process Data Flow (Sheet 7 of 13)

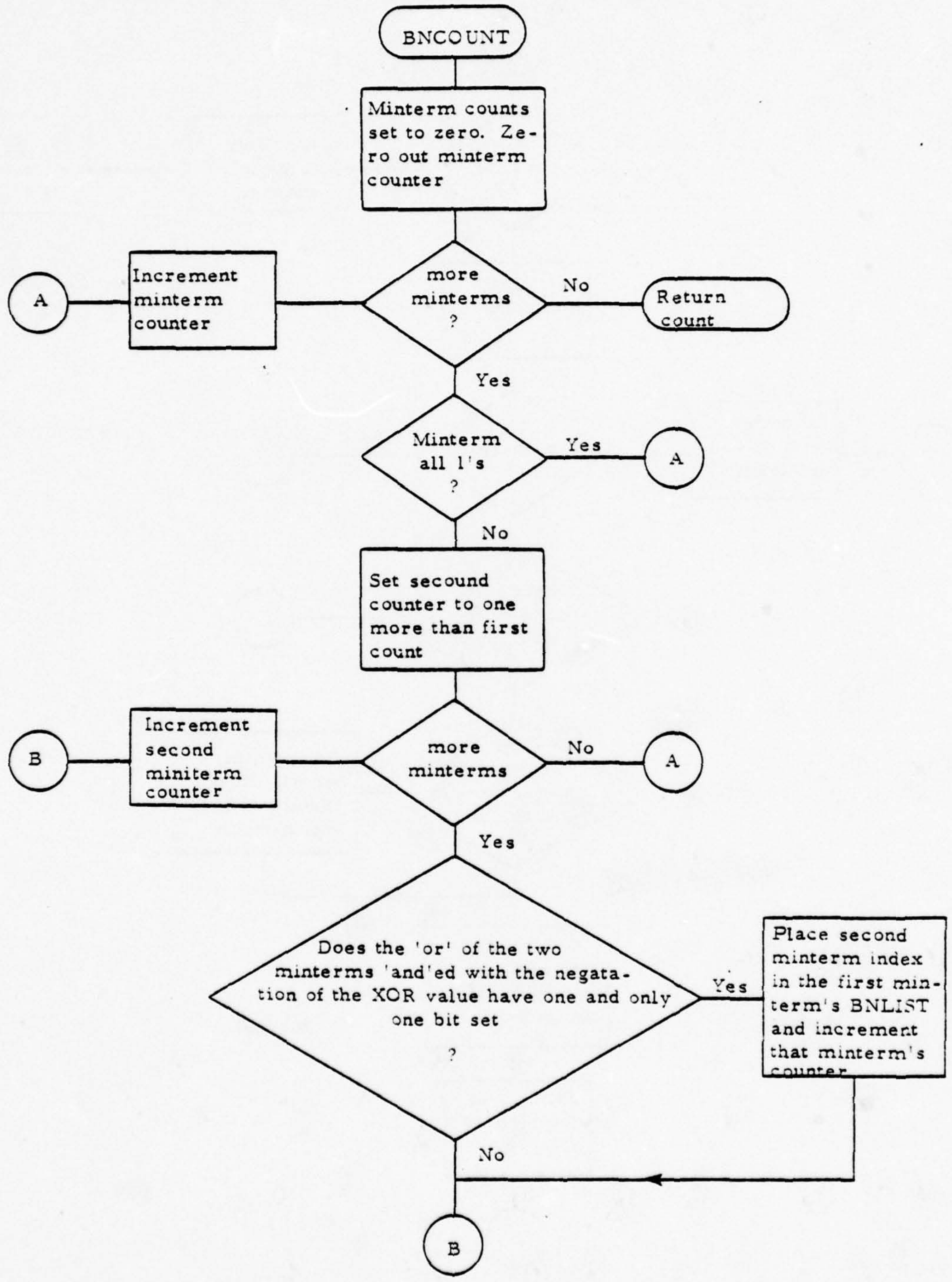

BNCOUNT
Minterm counts set to zero. Zero out minterm counter
more minterms ?
No
Return count
Yes
Increment minterm counter
A
Minterm all 1's ?
Yes
A
No
Set secound counter to one more than first count
more minterms
No
A
Yes
Increment second miniterm counter
B
Does the 'or' of the two minterms 'and'ed with the negatation of the XOR value have one and only one bit set ?
Yes
Place second minterm index in the first minterm's BNLIST and increment that minterm's counter
No
B

Figure 37. BEN Process Data Flow (Sheet 8 of 13)

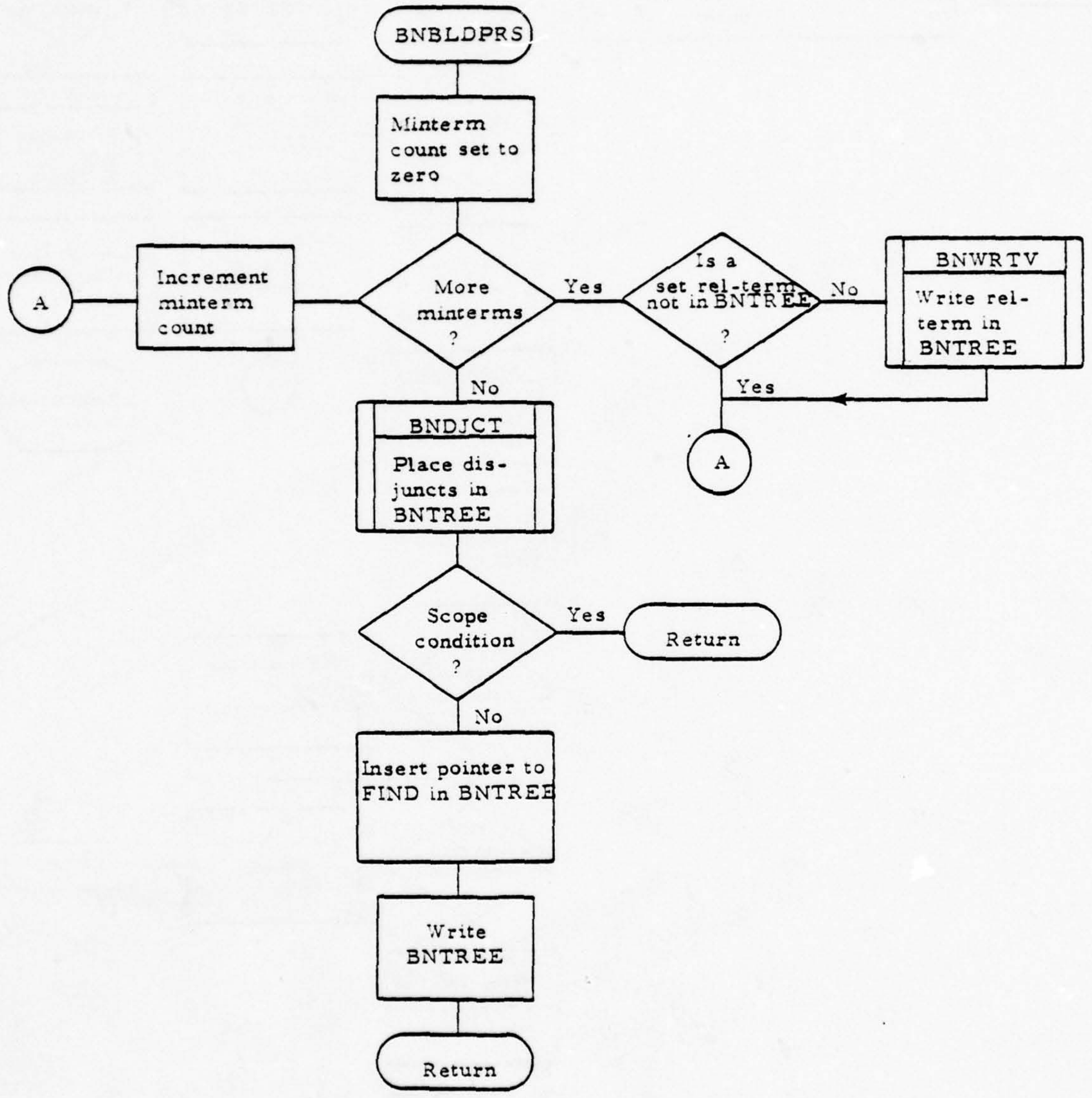


Figure 37. BEN Process Data Flow (Sheet 9 of 13)

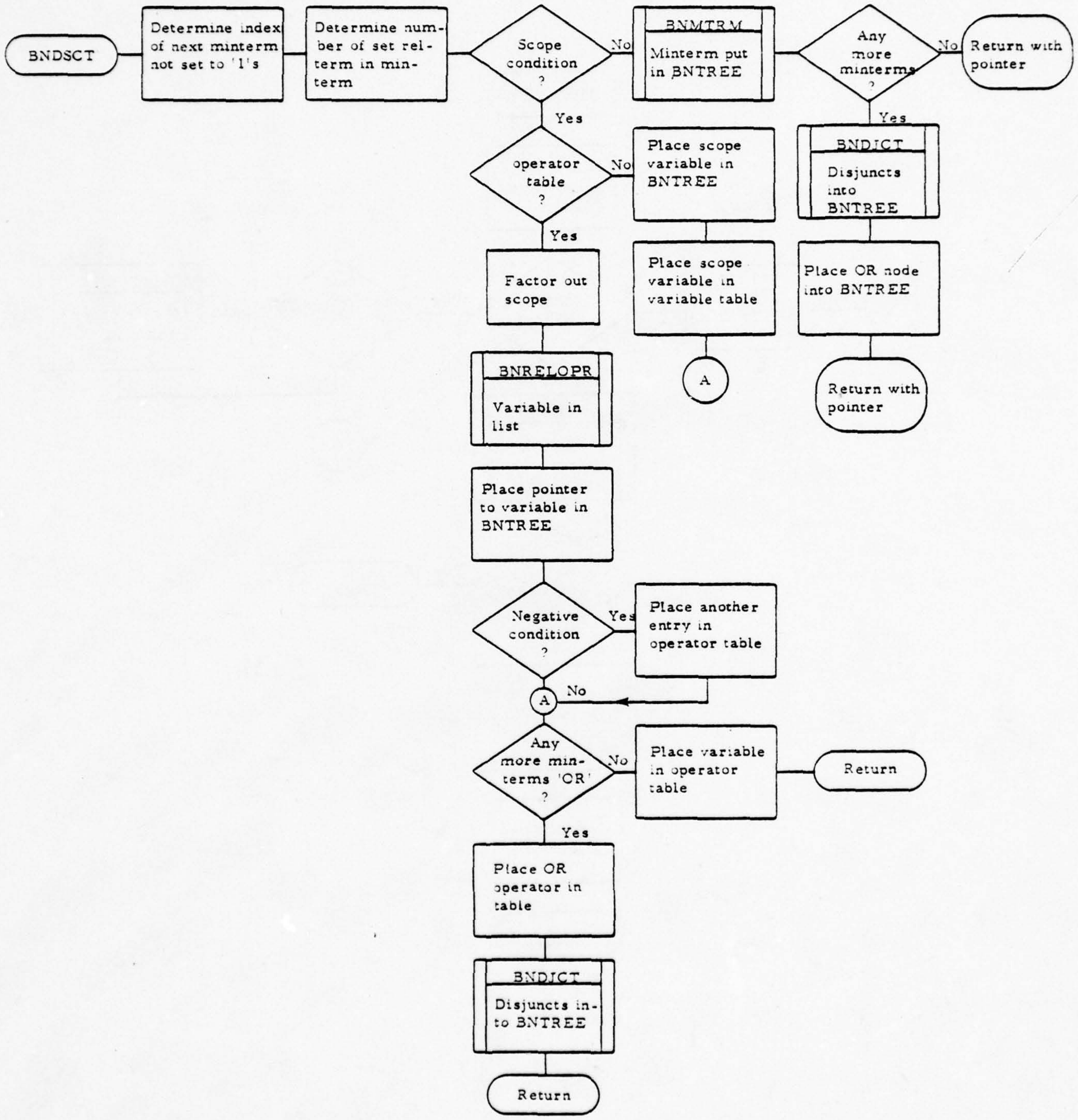

BNDSCT
Determine index of next minterm not set to '1's
Determine number of set relterm in minterm
Scope condition ?
No
BNMTRM
Minterm put in BNTREE
Any more minterms ?
No
Return with pointer
Yes
BNDICT
Disjuncts into BNTREE
Place OR node into BNTREE
Return with pointer
Yes
operator table ?
No
Place scope variable in BNTREE
Place scope variable in variable table
A
Yes
Factor out scope
BNRELOPR
Variable in list
Place pointer to variable in BNTREE
Negative condition ?
Yes
Place another entry in operator table
No
A
Any more minterms 'OR' ?
No
Place variable in operator table
Return
Yes
Place OR operator in table
BNDICT
Disjuncts into BNTREE
Return

Figure 37. BEN Process Data Flow (Sheet 10 of 13)

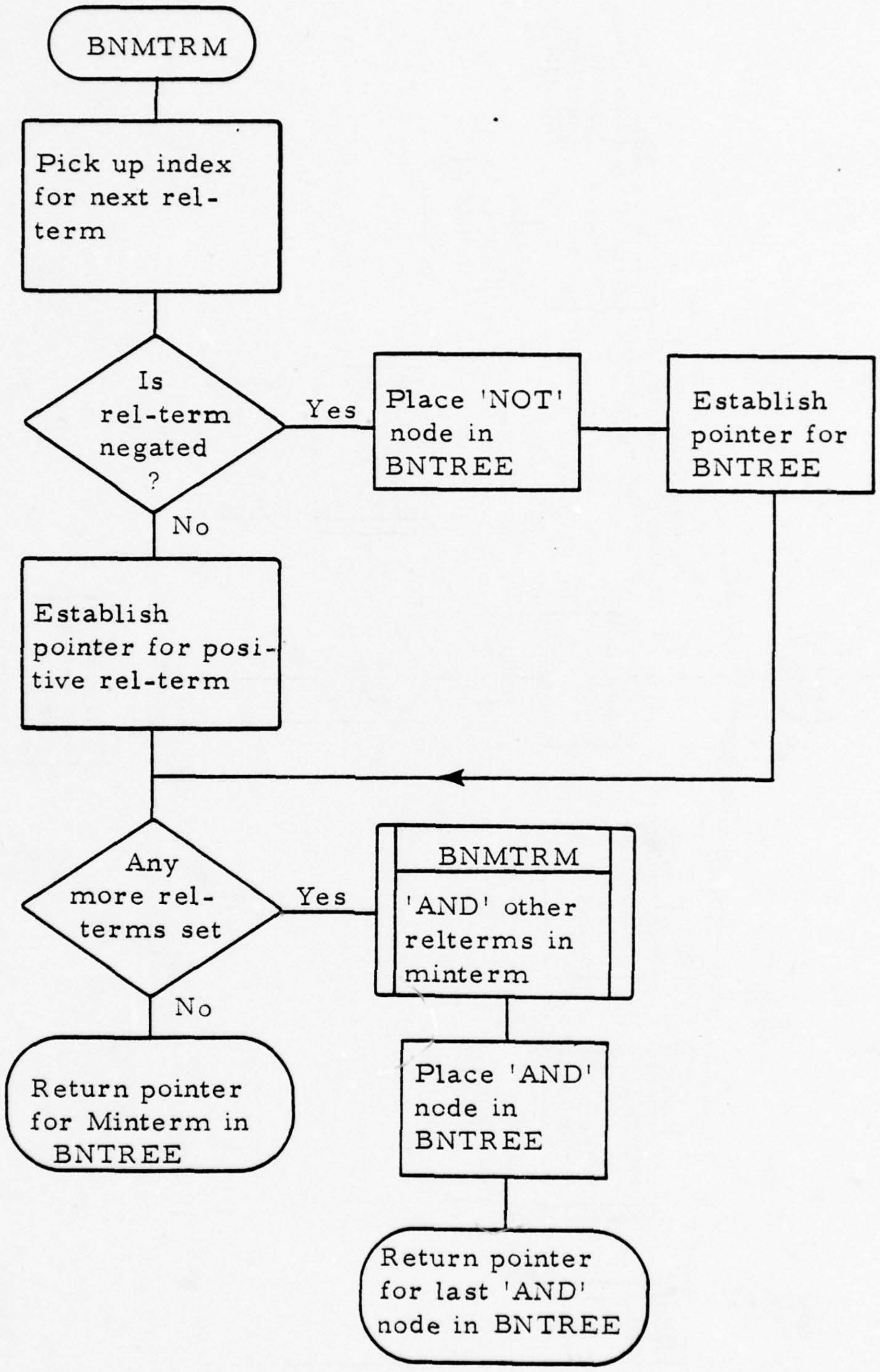

BNMTRM
Pick up index for next rel-term
Is rel-term negated ?
Yes
No
Place 'NOT' node in BNTREE
Establish pointer for BNTREE
Establish pointer for positive rel-term
Any more rel-terms set
Yes
No
BNMTRM
'AND' other relterms in minterm
Return pointer for Minterm in BNTREE
Place 'AND' node in BNTREE
Return pointer for last 'AND' node in BNTREE

Figure 37. BEN Process Data Flow (Sheet 11 of 13)

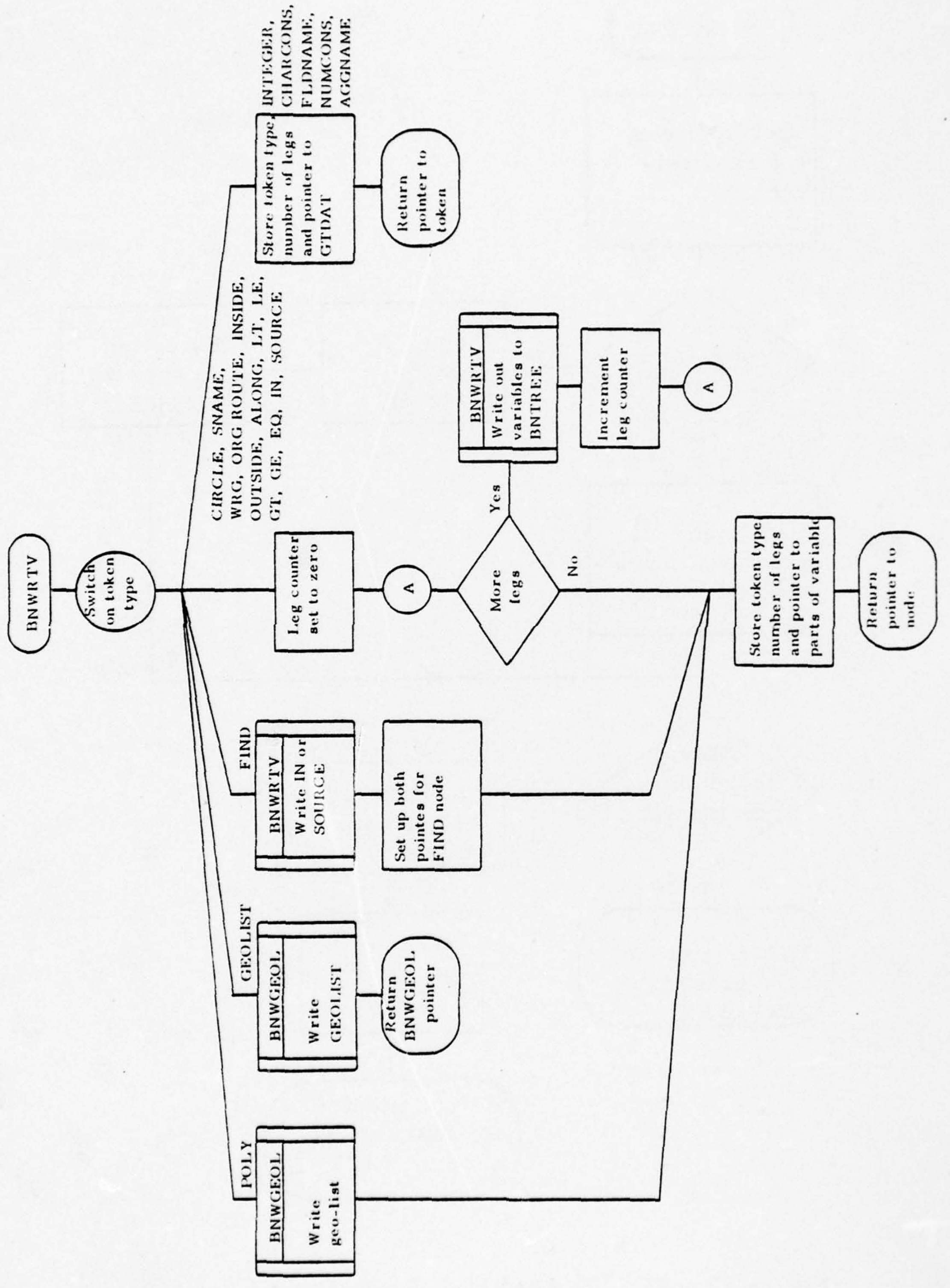

BNWRTV
Switch on token type
POLY
BNWGEOL
Write geo-list
GEOLIST
BNWGEOL
Write GEOLIST
Return BNWGEOL pointer
FIND
BNWRTV
Write IN or SOURCE
Set up both pointes for FIND node
CIRCLE, SNAME, WRG, ORG ROUTE, INSIDE, OUTSIDE, ALONG, LT, LE, GT, GE, EQ, IN, SOURCE
Leg counter set to zero
A
More legs
Yes
No
BNWRTV
Write out variables to BNTREE
Increment leg counter
A
Store token type number of legs and pointer to parts of variable
Return pointer to node
INTEGER, CHARCONS, FLDNAME, NUMCONS, AGGNAME
Store token type number of legs and pointer to GTDAT
Return pointer to token

Figure 37. BEN Process Data Flow (Sheet 12 of 13)

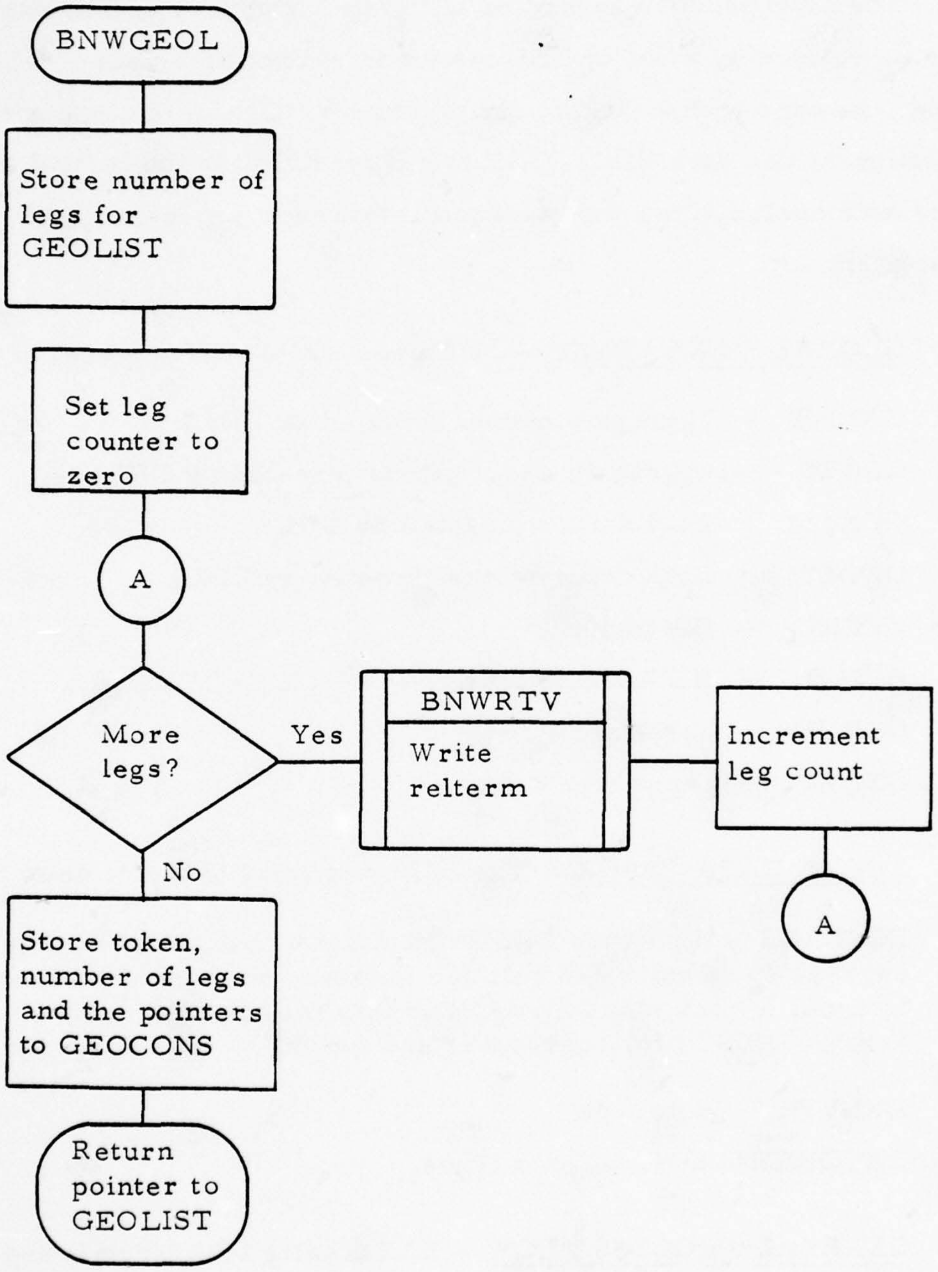

BNWGEOL
Store number of legs for GEOLIST
Set leg counter to zero
A
More legs?
Yes
BNWRTV
Write relterm
Increment leg count
A
No
Store token, number of legs and the pointers to GEOCONS
Return pointer to GEOLIST

Figure 37. BEN Process Data Flow (Sheet 13 of 13)

## DDL (Data Definition Language Process)

The DDL process generates UDL file dictionary definitions for a given file as defined by a set of DDL statements input by a user. DDL provides the user with a facility for defining a UDL file's composition for subsequent use with UDL. DDL is responsible for the lexical and syntactic analysis, as well as actual statement processing, of the DDL statement set.

GLOBAL DATA USAGE – DDL uses the following global data tables:

| | | |
|---|---|---|
| GANAM | – | aggregate names (created by DDL). |
| GADES | – | aggregate descriptions (created by DDL). |
| GFNAM | – | field names (created by DDL). |
| GFDES | – | field descriptions (created by DDL). |
| GFILN | – | file names. |
| GFILD | – | file descriptions. |
| GTREE | – | parse tree. |
| GTDAT | – | parse tree data. |

LOCAL DATA USAGE – The following local data are used by DDL:

DDSTACK – Structure DDSTACK follows the family lineages of aggregates to calculate relative indexes which point into the internal record map (GRMAP) or into the internal record data array (GRDAT) for aggregates and fields.

LXKYWRD–keywords.

LXTOKEN–keyword token types.

GENERAL PROCESS FLOW – EXEC calls DDL when it encounters either a SCHEMA statement or an EXECUTE SCHEMA statement. The EXECUTE SCHEMA statement provides users with the facility to store

DDL statements in an ordinary UNIX file, which can then be read by DDL for syntactic and semantic validation only, or for actual file dictionary generation. On the other hand, the encountering of a SCHEMA statement by EXEC implies an interactive environment with a user's terminal. If a UNIX file is being used, DDL calls the LEX/parse initialization (LXINIT) to process the first line (a SCHEMA statement) in the file. Henceforth, DDL handles both modes of input identically.

The SCHEMA statement must appear once in the file definition, and it must be the first line of the definition. Pertinent information, such as the filename and the target system mode of operation, is stored in the SCHEMA statement. The filename is authenticated to be a unique filename in GFILN. General target system information is conveyed through the key words LOCAL, REMOTE (BATCH) and REMOTE (INTER).

At the conclusion of processing any DDL statement, the LEX/parser is called upon to syntactically validate the next statement and to build the parse tree (GTREE) and the parse tree data (GTDAT).

It might be noted that throughout the DDL process an error message is printed whenever a syntactic or semantic user error is encountered. DDL attempts to catch all errors in the schema-definition-block: the error messages are output by the global function, GFERROR.

Following the SCHEMA statement is the schema-clause, a block of data structure definitions making up the file's logical structure. Within the block are two basic types of definitions: those defining fields and those defining aggregates. Aggregate-definitions are optional depending on whether a file has them, but a field-definition-block is mandatory. In other words, every file has a basic data set which contains at least one field. The basic data set definition is a sequence of field-definition statements and must occur first in the schema-clause.

A field-definition statement is indicated by the token type FIELD, followed by the field's name and a field-clause. This fieldname must be a unique name within the file. The mandatory field-clause specifies the field's type, its data-type, its access attributes, and its size. Two types of fields exist in UDL: single-valued (SV) and multivalued (MV). Following the field-type is the field's data-type specification. One of the following three data-types must be specified: character (CHAR), numeric (NUM), or geographic (GEO). Each field also has associated with it an interrogation attribute and a display attribute. Valid interrogation attributes are KEY, for fields which can be referenced in any FIND statement; NKEY (nonkeyed), for fields which cannot be referenced in FIND statements; and DEP (dependent) for fields which can only be referenced in dependent FIND statements. Two display attributes are recognized in UDL: visible (VIS) and invisible (INV). A field cannot logically be nonkeyed and invisible simultaneously.

The maximum field size (as an ASCII representation, regardless of the data-type) must be specified for each field in the form of an integer or a V (which symbolizes a variable length character string). A variable length character string is legal if the field is a visible, nonkeyed, and single-valued field whose data-type is character. If an integer is given for the field size and if the field is both single-valued and visible, the field's GIPTR is set to the parent aggregate GSIZE. The parent aggregate GSIZE is then increased by this field size.

Aggregates, when present in a file, are either arrays or repeating-groups, and must be specified following the basic data set definition block.

Subroutine DDAGG processes both types of aggregate-definition-blocks, distinguishing only when necessary. An array-definition-block begins with the token ARRAY; similarly, a repeating-group-definition-block begins with the token RGROUP. If a previous aggregate-definition existed, a check is made to ensure the proper closure of it with either an EARRAY or an ERGROUP. The aggregate statement is comprised of the aggregate name, its order (for arrays only) and an optional parent aggregate name. The aggregate name is confirmed to be either a unique name within the file or an undefined aggregate. At this point the array-statement specifies its order, which is defined to be the maximum number of occurrences allowed for the array. The final specification for both statement types is the name of another aggregate, the parent name. (The parent name is not necessarily defined at this point, thus relieving the user of defining aggregates in the restrictive parent-subordinate order.) The necessary family links (parent, subordinate, and sibling) are made to give shape to a hierarchical structure. If no parent name is specified, the parent is the basic data set.

The array-definition-block must be terminated by the token EARRAY; the repeating-group-definition by ERGROUP. Upon the recognition of these tokens, two checks must be made. The first check is to validate that an aggregate definition does indeed exist. If it does, verification is made that at least one field is defined in the aggregate-definition-block.

The final statement in a schema-definition is ESCHEMA. Upon recognition of the ESCHEMA token, the DDL process makes a final check and, if it finds no preceding errors, writes out the pertinent files.

The final pass checks for the following errors:

a. Improper closure of the last aggregate-definition-block.

b. Undefined aggregates.

c. An aggregate which directly or indirectly (through other aggregates) is its own parent.

d. Repeating-groups whose parents are arrays (arrays cannot be parents of repeating-groups).

The relative indexes which point into the internal record map (GRMAP) or into the internal record data array (GRDAT) are calculated and stored. The pushdown-popup stack (DDSTACK) is used to follow the lineages of aggregates, starting with the basic data set. An aggregate's subordinates are assigned their GMPTRs in the consecutive order of their input (as indicated by sibling links). Having assigned the subordinates their pointers, the aggregate's multivalued and single-valued variable length fields (which are visible) are now assigned their GIPTRs.

A schema-definition is not a valid definition if any errors were detected throughout the processing of the schema-definition. If any errors were noted or the mode is validation only, control is returned to EXEC. If, however, the schema-definition is error free, DDL writes out the file's dictionary definitions. These files include the information concerning the aggregates (GANAM and GADES) and the fields (GFNAM and GFDES). Finally, DDL stores the filename and description in the file name and file description (GFILN and GFILD) tables and returns control to EXEC. See figure 38 for process data flow.

## MAJOR FUNCTION DESCRIPTIONS

LXINIT (LEX/Parse Initialization) – LXINIT is merely an initialization function for the LEX/parse portion of this process. It initializes pertinent data including a flag which it returns to the calling function to indicate end-of-file (EOF), syntax error, or statement accepted. LXINIT calls the parsing function (YYPARSE) and returns with the flag value.

<u>YYPARSE (Parser)</u> – YYPARSE is a parser which is produced by the Yet Another Compiler-Compiler (YACC) program running under UNIX. YACC produces the parser as well as a set of tables which the parser uses to organize the tokens passed to it by the lexical analyzer (YYLEX). These tables reflect the grammar of the DDL statements. YYPARSE calls YYLEX, which returns a value called a token type. If the token type is invalid according to the input statement syntax rules, an error message is output and YYPARSE calls YYLEX continuously until an end-of-input token type is returned. YYPARSE then returns with the return flag set to reflect a syntax error. If the token type is valid according to the input rules, the action specified in the YACC run is performed. The actions that can be invoked are: no action, build a parse tree node with the specified number of legs, store the final token type and write out the parse tree and parse tree data. After the action is performed, YYPARSE calls YYLEX for the next token type. When the end-of-input token type is received and its action performed, YYPARSE returns with the return flag set to reflect statement acceptance. If YYLEX encounters an EOF, it returns an end-of-input token type to YYPARSE and sets the return flag to reflect EOF.

<u>YYLEX (Lexical Analyzer)</u> – YYLEX is the function which actually inputs and processes each DDL statement. YYLEX reads and saves each input character until a delimiter is encountered. The delimiters for ADAPT I DDL are: comma, newline (carriage return), space, left parenthesis, right parenthesis and semi-colon. If an EOF is encountered, the return flag is set to reflect EOF and an end-of-input token type is returned. If a delimiter is encountered by itself, it is either passed to the parser (comma, left or right parenthesis, or semi-colon), or it is ignored.

When a delimiter terminates a string of characters, YYLEX must determine what the characters represent. If the characters form a DDL keyword, YYLEX returns with that keyword's token type. Otherwise, YYLEX determines the type of constant which is formed by the characters (name, integer), stores the value of the constant in the parse tree data (GTDAT), stores a terminal node in the parse tree (GTREE), and returns with the constant's token type. If YYLEX encounters an error (e. g. , too many characters input), it returns an invalid token type to the parser and thereby generates a syntax error message.

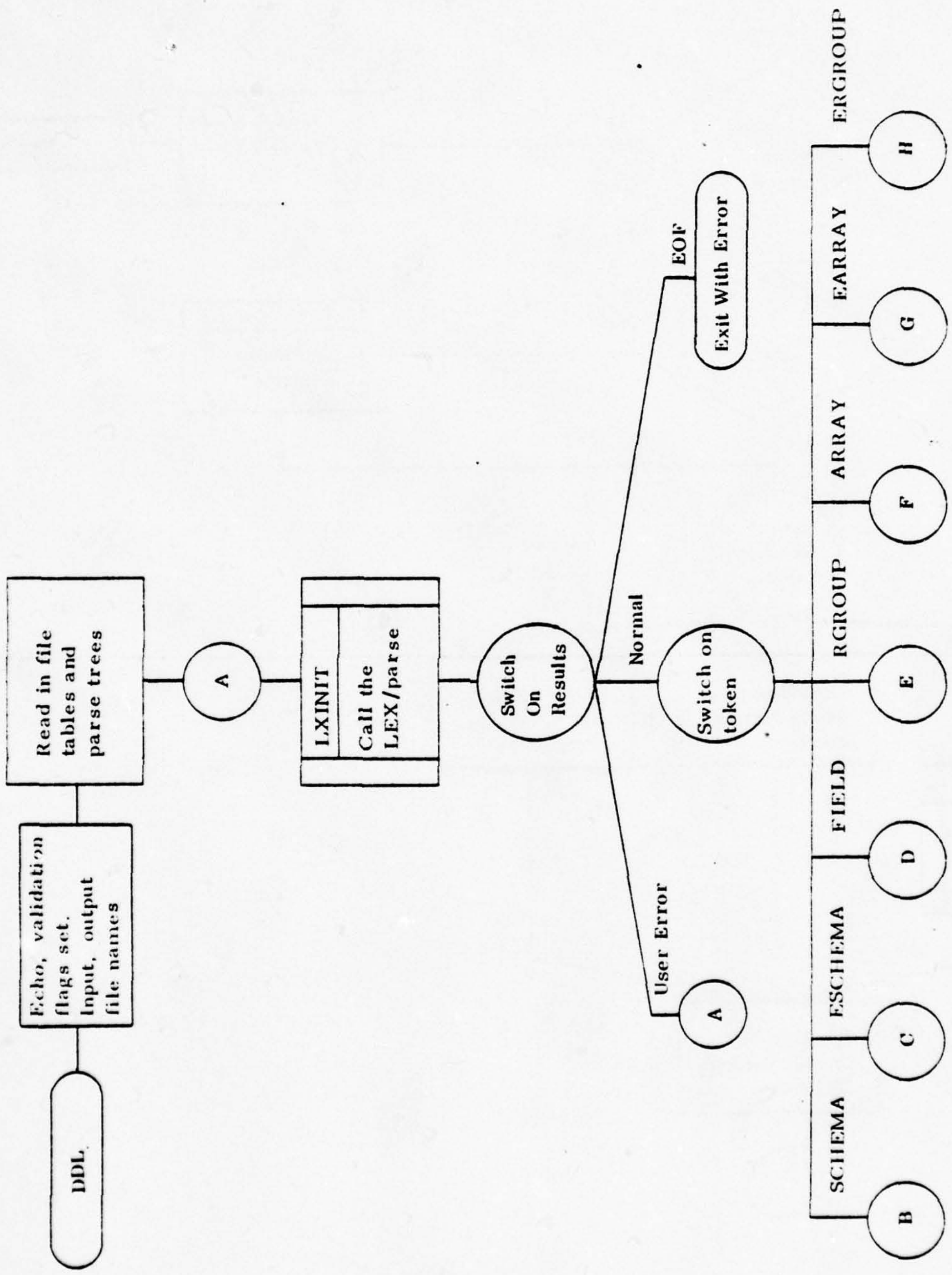


Figure 38. DDL Process Data Flow (Sheet 1 of 9)

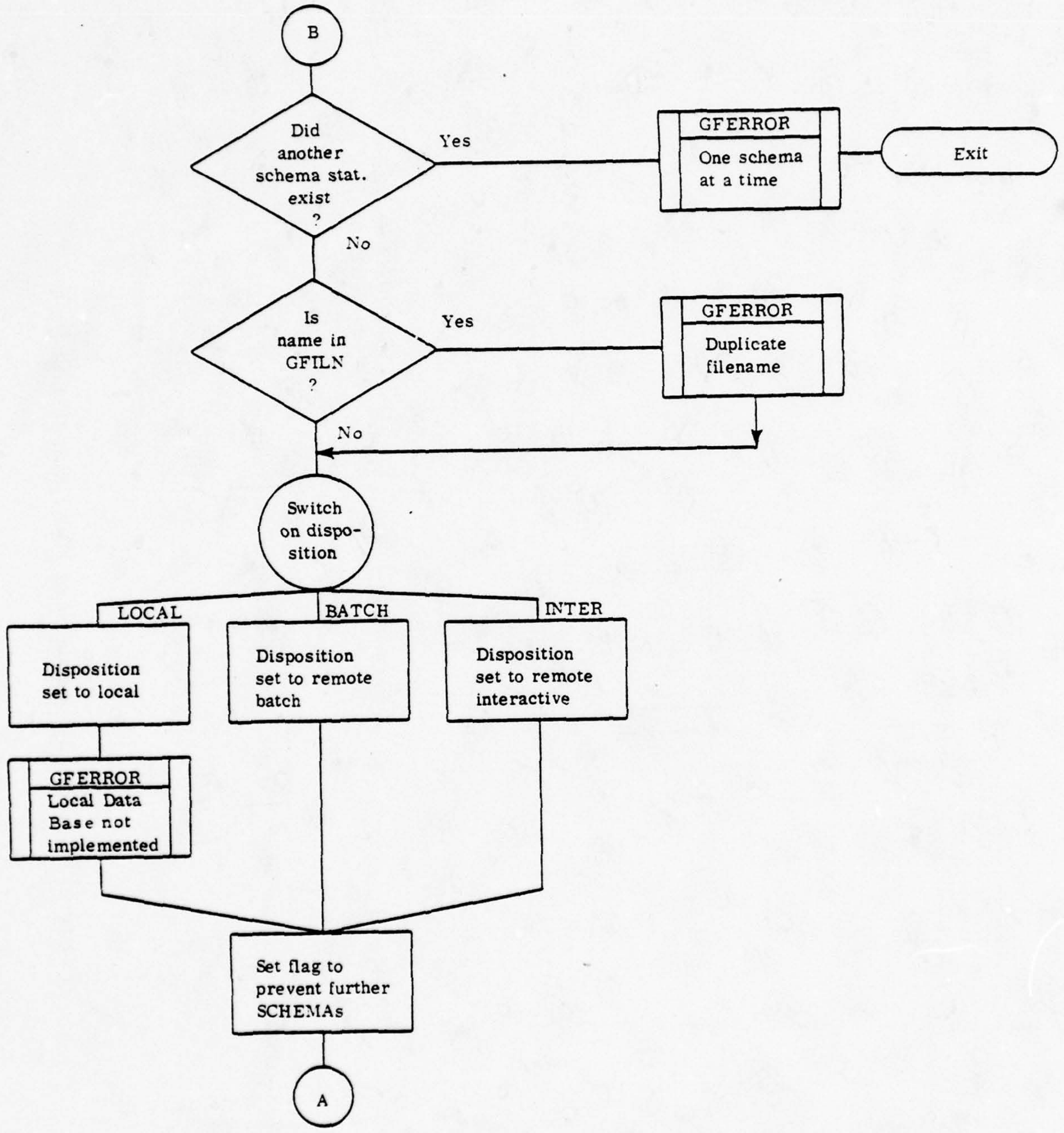

B
Did another schema stat. exist ?
Yes
No
GFERROR
One schema at a time
Exit
Is name in GFILN ?
Yes
No
GFERROR
Duplicate filename
Switch on disposition
LOCAL
BATCH
INTER
Disposition set to local
Disposition set to remote batch
Disposition set to remote interactive
GFERROR
Local Data Base not implemented
Set flag to prevent further SCHEMAs
A

Figure 38. DDL Process Data Flow (Sheet 2 of 9)

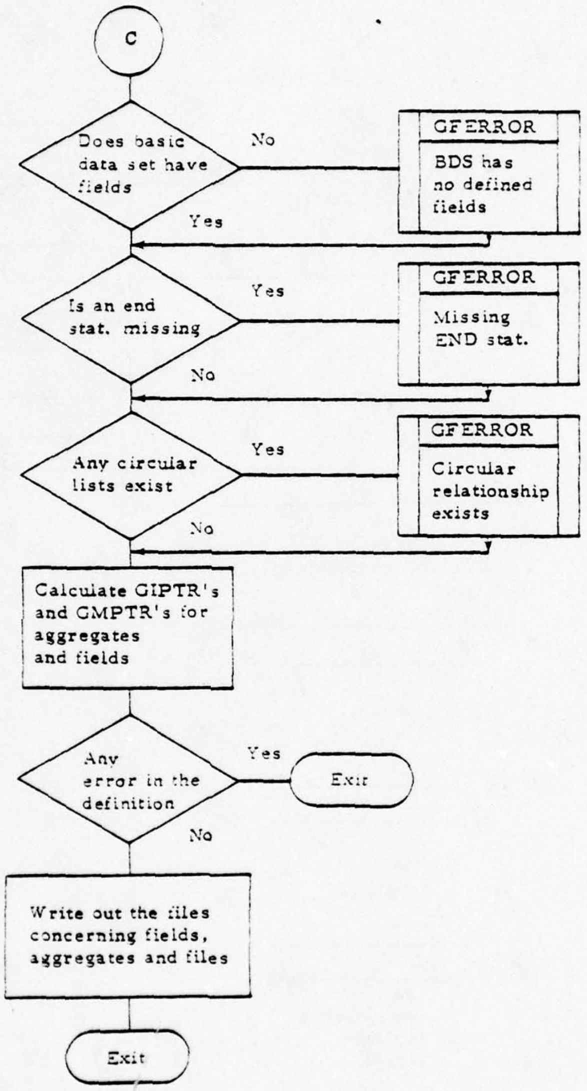

C
Does basic data set have fields
No
Yes
GFERROR
BDS has no defined fields
Is an end stat. missing
Yes
No
GFERROR
Missing END stat.
Any circular lists exist
Yes
No
GFERROR
Circular relationship exists
Calculate GIPTR's and GMPTR's for aggregates and fields
Any error in the definition
Yes
Exit
No
Write out the files concerning fields, aggregates and files
Exit

Figure 38. DDL Process Data Flow (Sheet 3 of 9)

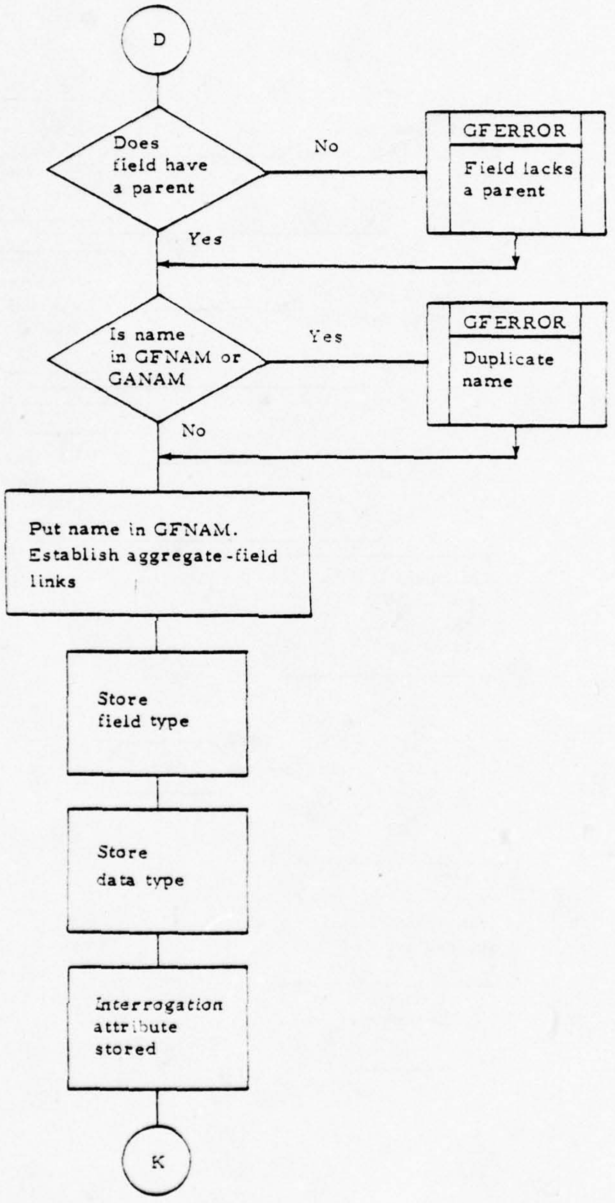


Figure 38. DDL Process Data Flow (Sheet 4 of 9)

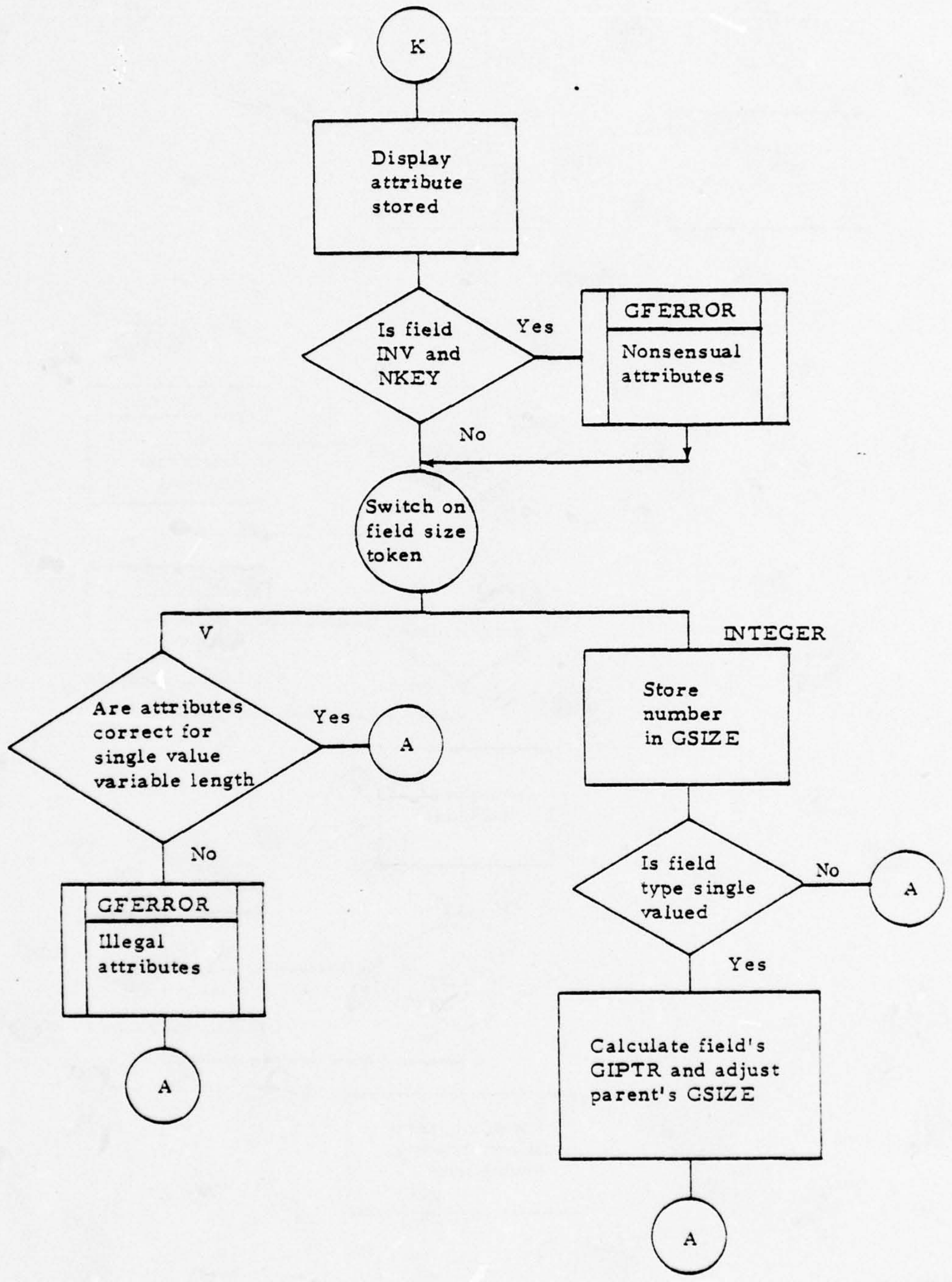

K
Display attribute stored
Is field INV and NKEY
Yes
GFERROR
Nonsensual attributes
No
Switch on field size token
V
INTEGER
Are attributes correct for single value variable length
Yes
A
No
GFERROR
Illegal attributes
A
Store number in GSIZE
Is field type single valued
No
A
Yes
Calculate field's GIPTR and adjust parent's GSIZE
A

Figure 38. DDL Process Data Flow (Sheet 5 of 9)

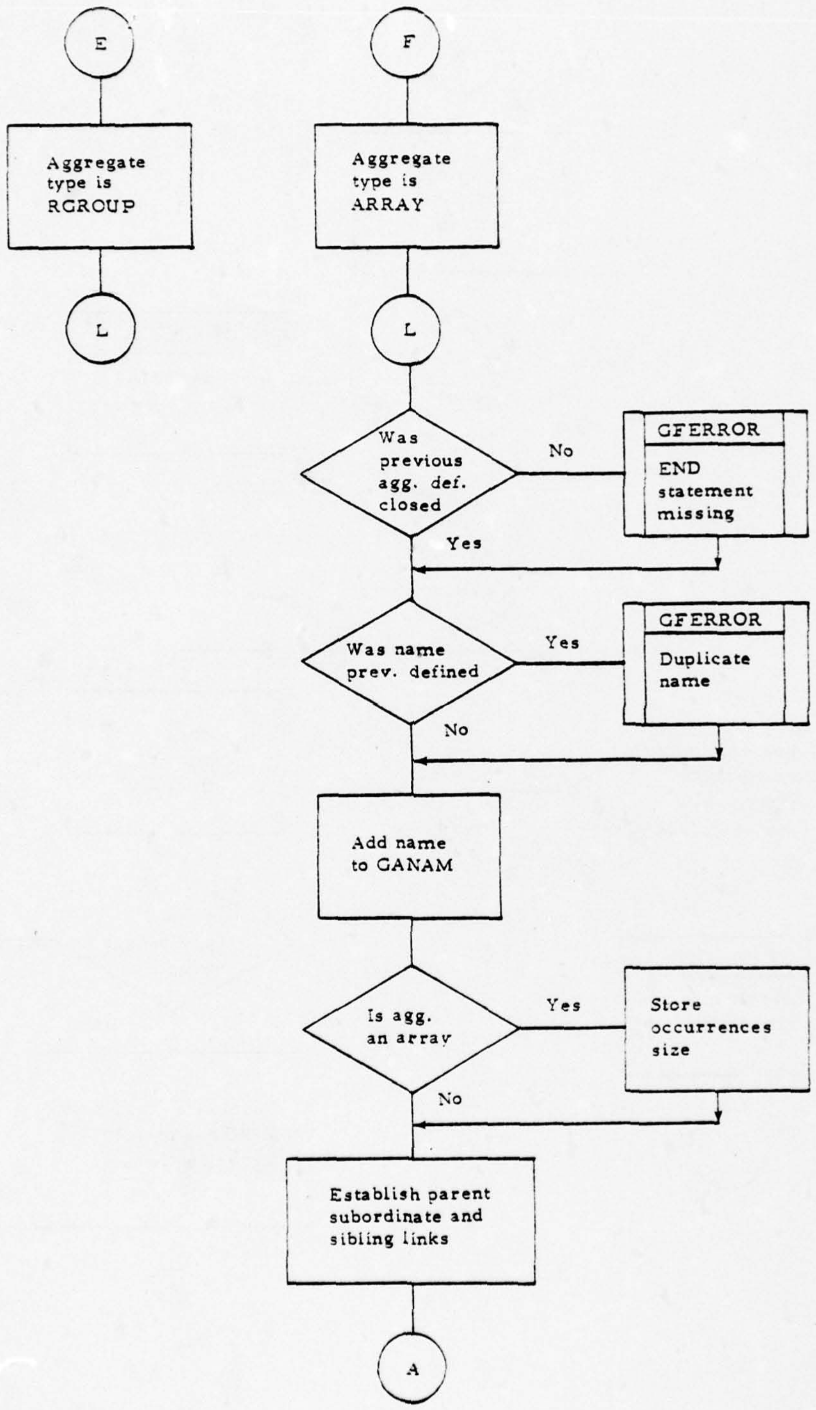


Figure 38. DDL Process Data Flow (Sheet 6 of 9)

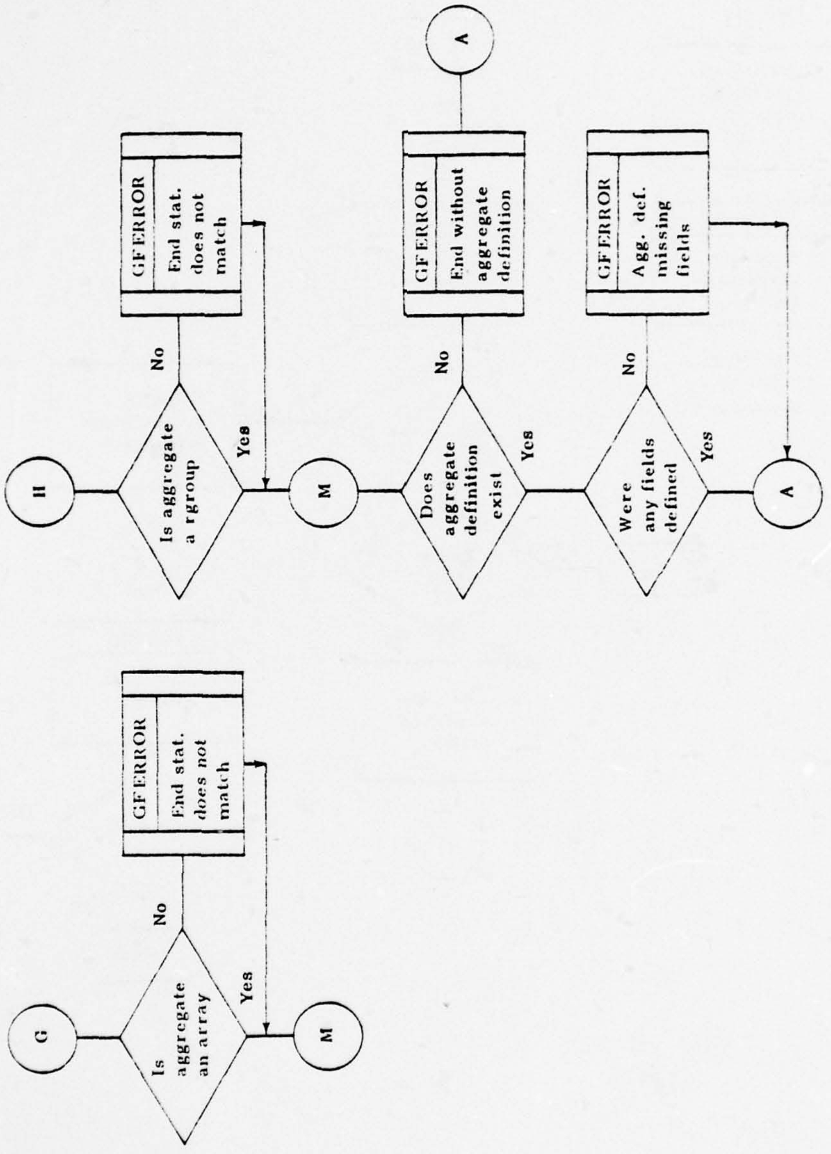


Figure 38. DDL Process Data Flow (Sheet 7 of 9)

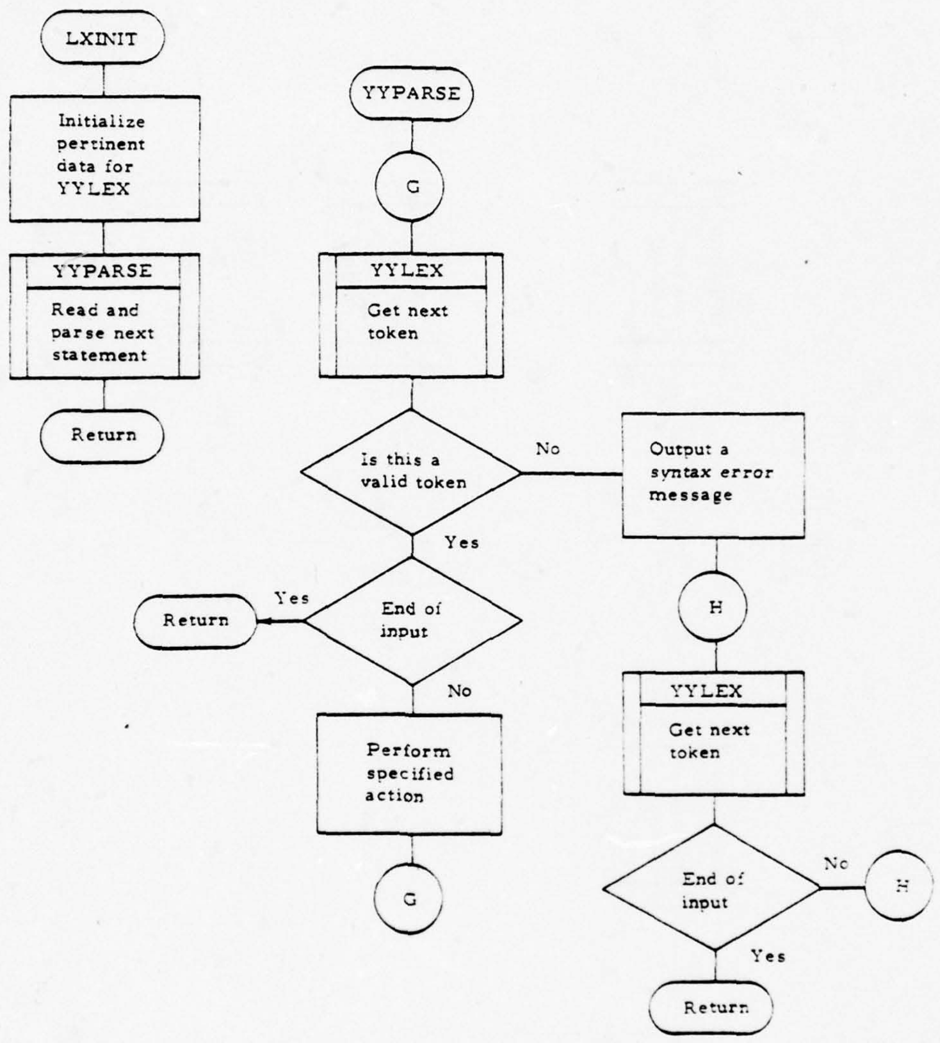


Figure 38. DDL Process Data Flow (Sheet 8 of 9)

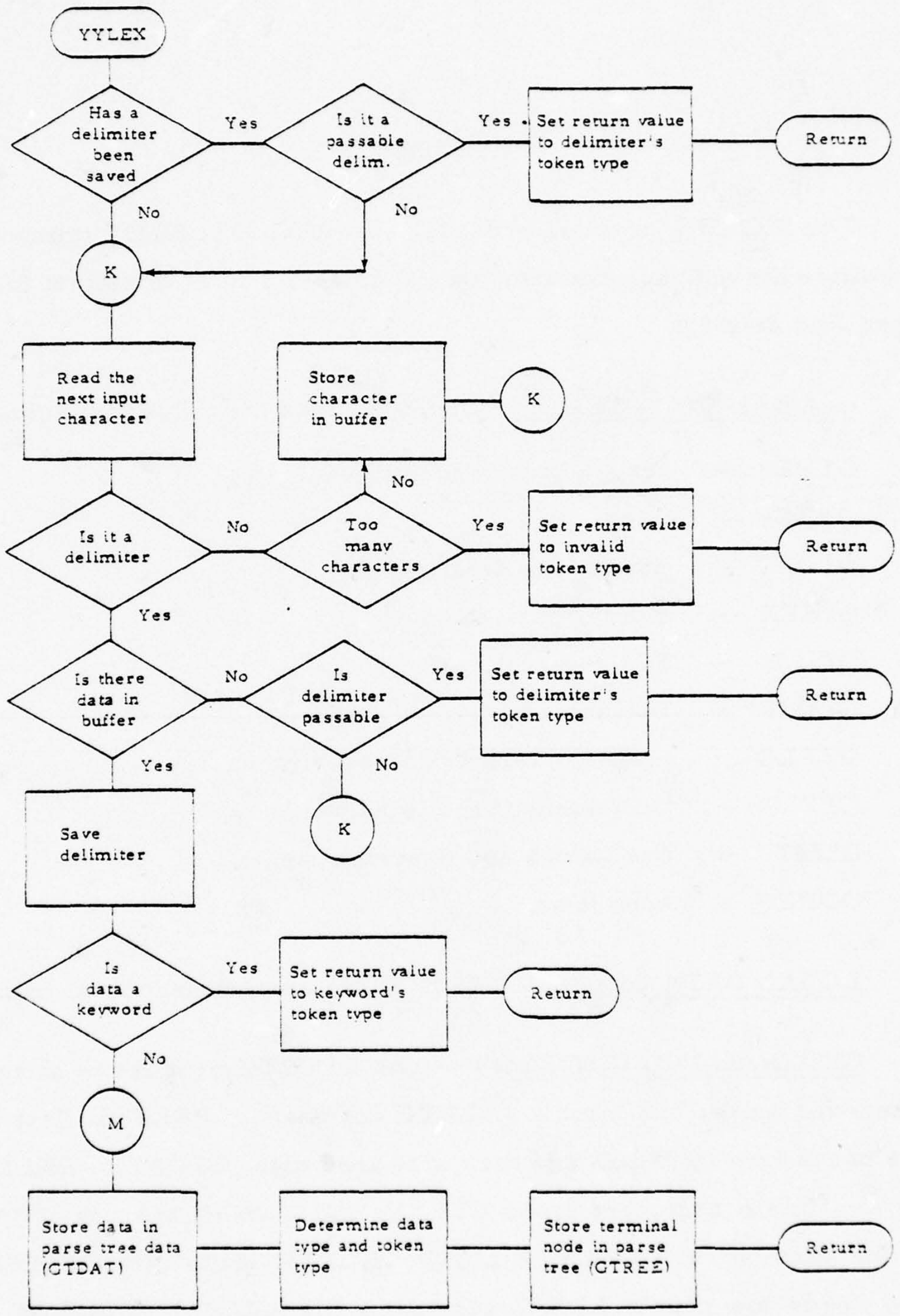

YYLEX
Has a delimiter been saved
Yes
No
Is it a passable delim.
Yes
No
Set return value to delimiter's token type
Return
K
Read the next input character
Store character in buffer
K
No
Is it a delimiter
No
Too many characters
Yes
Set return value to invalid token type
Return
Yes
Is there data in buffer
No
Is delimiter passable
Yes
Set return value to delimiter's token type
Return
Yes
No
K
Save delimiter
Is data a keyword
Yes
Set return value to keyword's token type
Return
No
M
Store data in parse tree data (GTDAT)
Determine data type and token type
Store terminal node in parse tree (GTREE)
Return

Figure 38. DDL Process Data Flow (Sheet 9 of 9)

## DELETE

The DELETE process processes the UDL DELETE command. It modifies the appropriate data files and deletes files to reflect the specified deletion.

GLOBAL DATA USAGE – DELETE uses the following global data:

| | |
|---|---|
| GTDES – | transaction descriptions. |
| GTREE – | parse tree. |
| GTDAT – | parse tree data. |
| GUSER – | user descriptions. |
| GFILD – | file descriptions. |
| GFILN – | file names. |
| GTFLD – | transformation file description. |
| GTFLN – | transformation file names. |
| GLIST – | list names and descriptions. |
| GOPEN – | open files. |

LOCAL DATA USAGE – DELETE has no pertinent local data.

GENERAL PROCESS FLOW – The DELETE process is activated whenever a user has input a DELETE command. DELETE first reads the parse tree (GTREE) and the parse tree data (GTDAT). DELETE picks up the delete argument from GTREE: LIST, SCHEMA, or TRANFILE. If the request is a DELETE LIST, DELETE locks (via GFLOCK) and reads the user's List Descriptions file (GLIST). DELETE validates the list name in GTDAT, and if the name is invalid, DELETE calls DERROR to output an error and exit. If the list status is neither completed nor displayed, then the list cannot be deleted, and DELETE calls DERROR to output an error and exit. Otherwise, DEDELST is called to delete the list and DERROR is called to output a final message and exit.

For a DELETE SCHEMA or DELETE TRANFILE, DELETE validates that the user is the ADAPT superuser. If not, DERROR is called to output an error and exit. DELETE locks (via GFLOCK) and reads the User Descriptions file (GUSER), and checks user statuses to determine whether any other users are logged onto ADAPT. If so, DERROR is called to output an error and exit.

Next, the File Names file (GFILN) is read and DELETE validates the file name in GTDAT. If the file name is invalid, DERROR is called to output an error and exit. The Transformation File Descriptions file (GTFLD) is read next. If the specified file has a tranfile definition, a flag is set. If there is no tranfile definition and the request is DELETE TRANFILE, then DERROR is called to output an error and exit. DELETE locks (via GFLOCK) and reads the Transaction Descriptions file (GTDES) and determines whether there are any transactions. If there are transactions, then for each transaction, the GTDES item is read and, if the referenced file is the deleted one and the status is active, the transaction's job is cancelled by DEBQRD and the transaction is deleted by DEDTRAN.

For users registered in ADAPT, DELETE next processes each user's list descriptions file (GLIST). For each user, DELETE locks (via GFLOCK) and reads that user's GLIST file. If the user has any lists, for each list, the following tests and actions are made. If the list is for the specified file, and either the file's schema is being deleted or the list status is active, then the list is deleted by DEDELST. The GLIST file is then written back to disk and unlocked via GFUNLOCK.

DELETE unlocks GTDES via GFUNLOCK after processing all users' GLIST files. Next, the File Descriptions file (GFILD) is read. If a tranfile is to be deleted, either explicitly or by a file's schema deletion, then the file's transformation data files (GPFDS, GTADS, GTFDS, GTNAM) are unlinked, zeroes are written to the file's transformation description in GTFLD, and blanks are written to the file's transformation name in GTFLN. If a file's schema is to be deleted, then the file's definition files (GADES, GFDES, GANAM, GFNAM) are unlinked, the file's description is deleted from GFILD, and the file name is deleted from GFILN. For either a TRANFILE or SCHEMA deletion, DELETE reads the superuser's Open Files file (GOPEN). If the specified file had been opened by a previous OPEN command, then that open file is set to zero and GOPEN is written out. DERROR is called to output a final message and exit. See Figure 39 for data flow.

## MAJOR FUNCTION DESCRIPTIONS

DERROR (Output Message and/or Exit) – DERROR is called with an error/message number or zero, and an exit flag set to either zero or non-zero. If the error/message flag is set, then the specified message is output via GFERROR. If the exit flag is not set, then DERROR returns. Otherwise, DERROR unlocks any locked files via GFUNLOCK and then exits.

DEDELST (Delete a List) – DEDELST is called with a list ID of the list to be deleted. DEDELST deletes the list's description from GLIST and unlinks the list's Record Map (GRMAP) and Record Data (GRDAT) files. If the request is a DELETE LIST, then GLIST is written out to disk, GLIST is unlocked via GFUNLOCK, and GTDES is locked via GFLOCK. DEDELST then opens GTDES and reads items until the list's transaction is found. If the list data had not been displayed, then DEBQRD is called to cancel the transaction's job(s). DEDTRAN is called to delete the transaction, GTDES is closed and DEDELST returns.

DEDTRAN (Delete Transaction) – DEDTRAN unlinks the five possible transaction files: GTREE and GTDAT for the DISPLAY or SAVE statement, GPFDS for position-oriented network responses, transaction output, and network response. The transaction description item in GTDES is set to zeroes and written out to disk. If the transaction consisted of multiple queries (multi-INTGs), each item in GTDES which related to the transaction is also set to zeroes and written out. The first item of GTDES, which contains the total number of transactions in ADAPT, is read, the transaction count is decremented, and the GTDES item is written back out. If the request is DELETE LIST, then GTDES is unlocked via GFUNLOCK and GUSER is locked via GFLOCK. DEDTRAN opens and reads GUSER. The transaction count for the user who had instigated the deleted transaction is decremented and GUSER is written out to disk. GUSER is closed and DEDTRAN returns.

DEBQRD (Cancel Transaction) – DEBQRD performs a fork/execute sequence in order to call BQRD (the TAS Batch Query and Response Dispatcher). BQRD is called with a function code of CANCEL, the job's application ID, and the JOBID of the batch query (INTG) to be cancelled. If the transaction consisted of multiple queries (multi-INTGs), then BQRD is repeatedly called for each batch query in the transaction. After all batch queries relating to the transaction have been cancelled, DEBQRD returns.

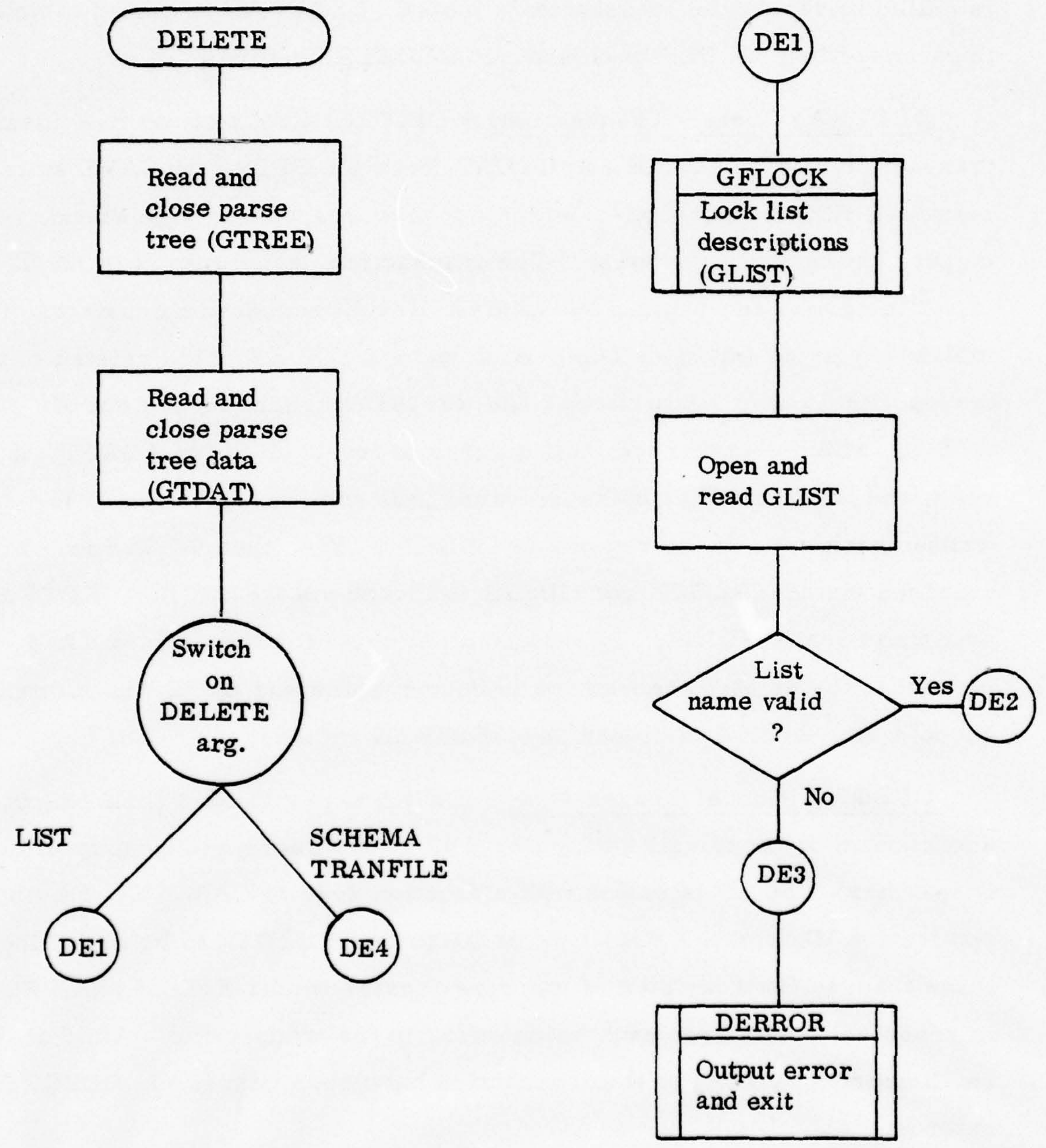


Figure 39. DELETE Process Data Flow (Sheet 1 of 12)

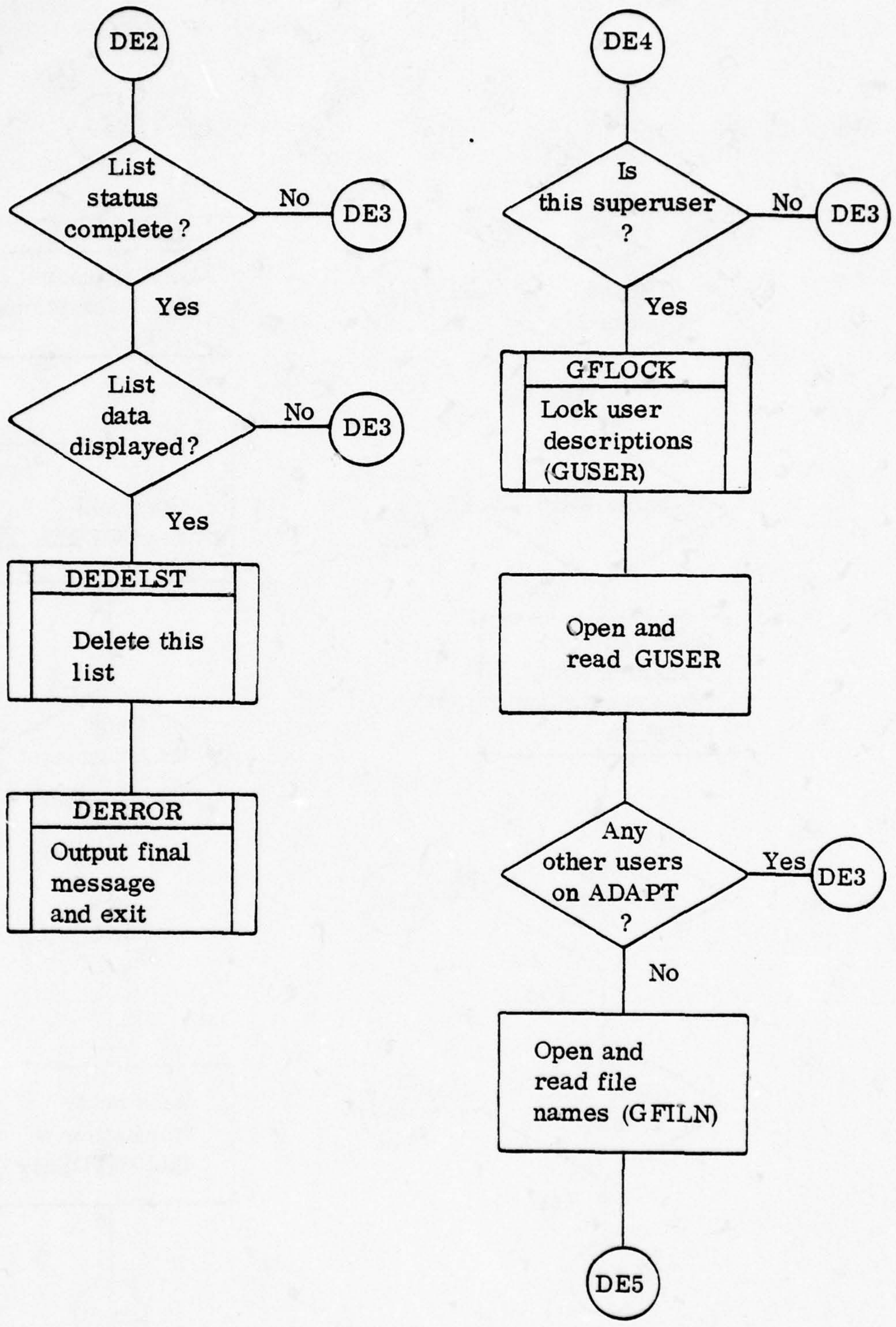


Figure 39. DELETE Process Data Flow (Sheet 2 of 12)

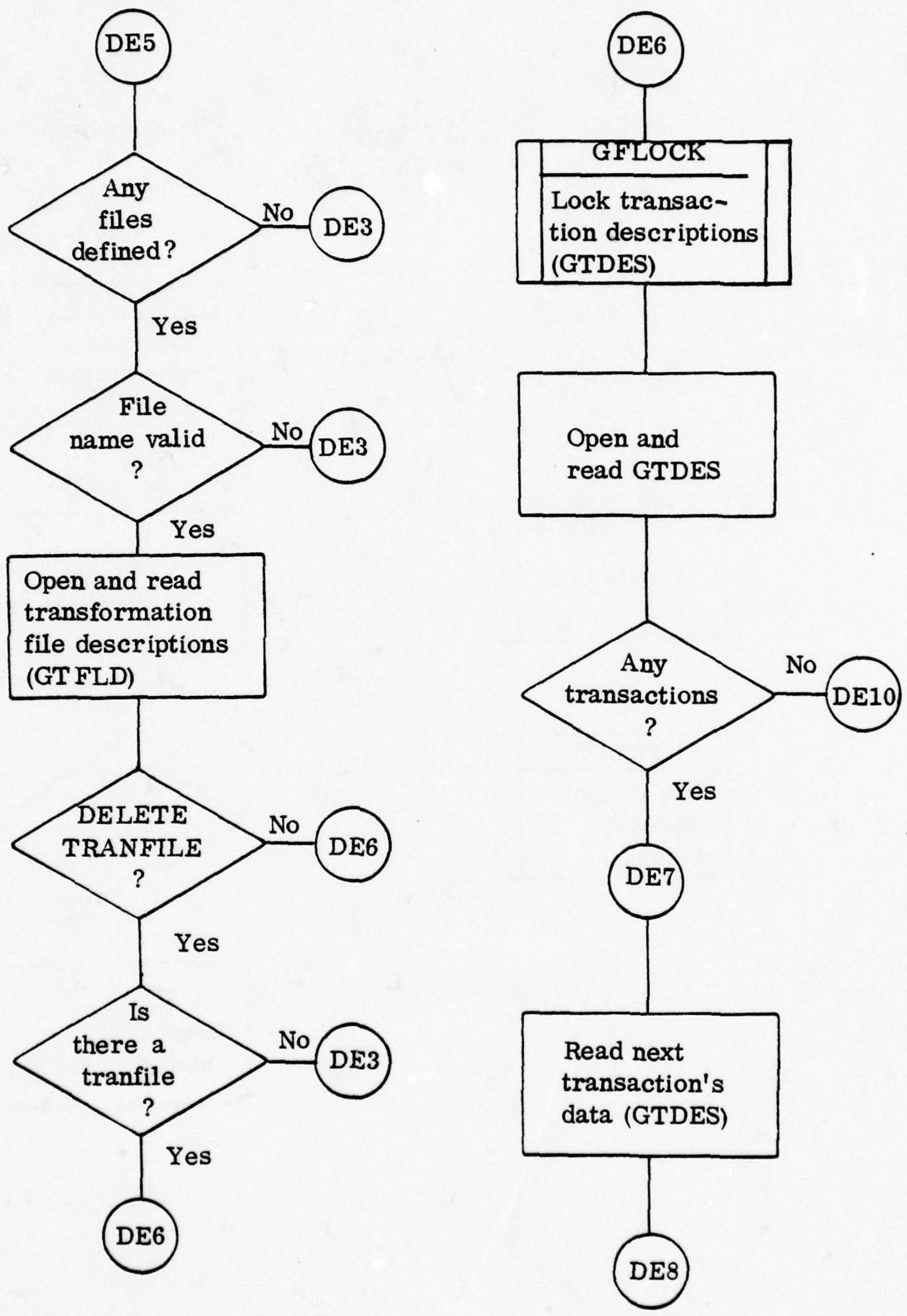


Figure 39. DELETE Process Data Flow (Sheet 3 of 12)

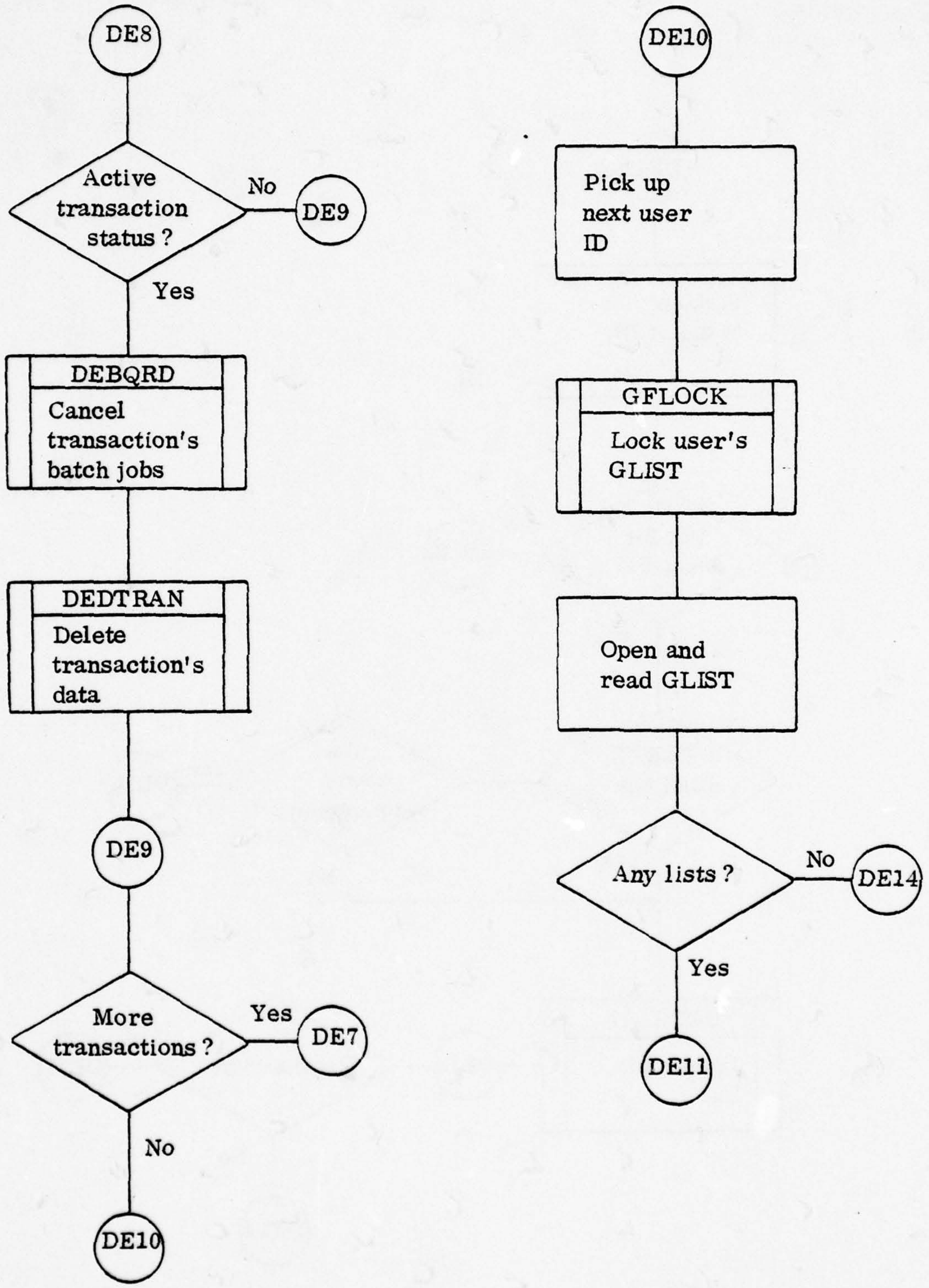


Figure 39. DELETE Process Data Flow (Sheet 4 of 12)

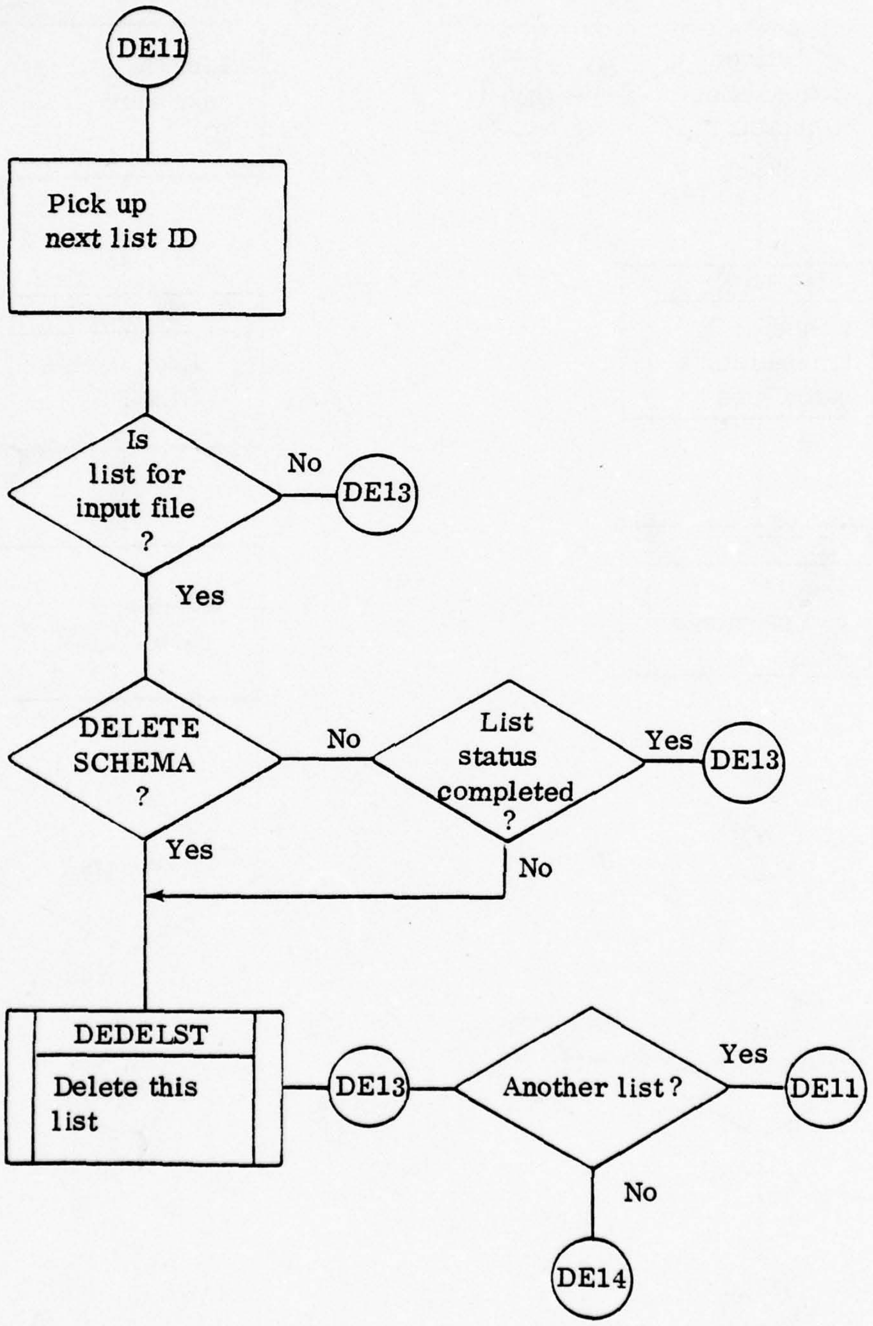


Figure 39. DELETE Process Data Flow (Sheet 5 of 12)

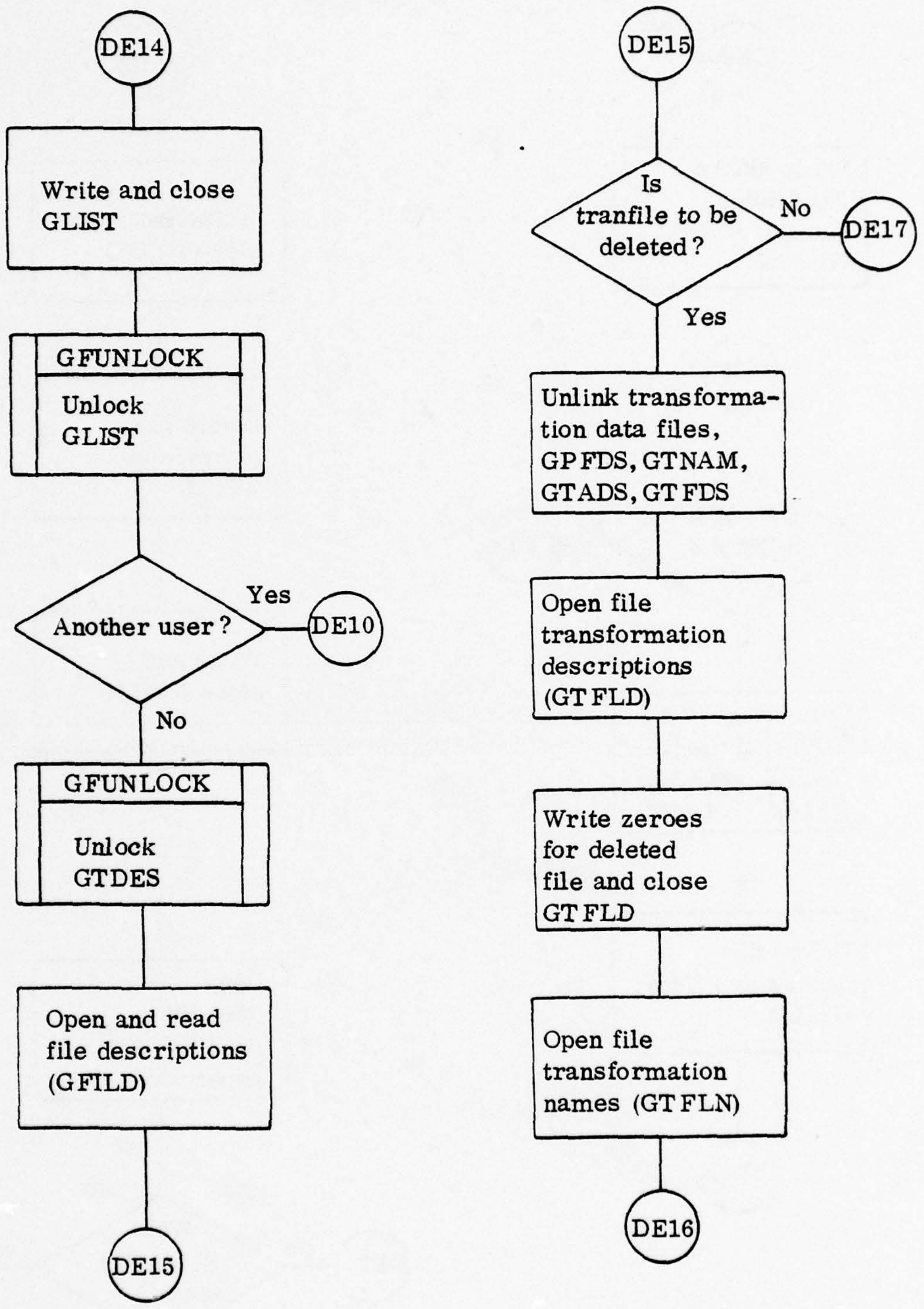

DE14
Write and close GLIST
GFUNLOCK
Unlock GLIST
Another user?
Yes
DE10
No
GFUNLOCK
Unlock GTDES
Open and read file descriptions (GFILD)
DE15
DE15
Is tranfile to be deleted?
No
DE17
Yes
Unlink transformation data files, GPFDS, GTNAM, GTADS, GTFDS
Open file transformation descriptions (GTFLD)
Write zeroes for deleted file and close GTFLD
Open file transformation names (GTFLN)
DE16

Figure 39. DELETE Process Data Flow (Sheet 6 of 12)

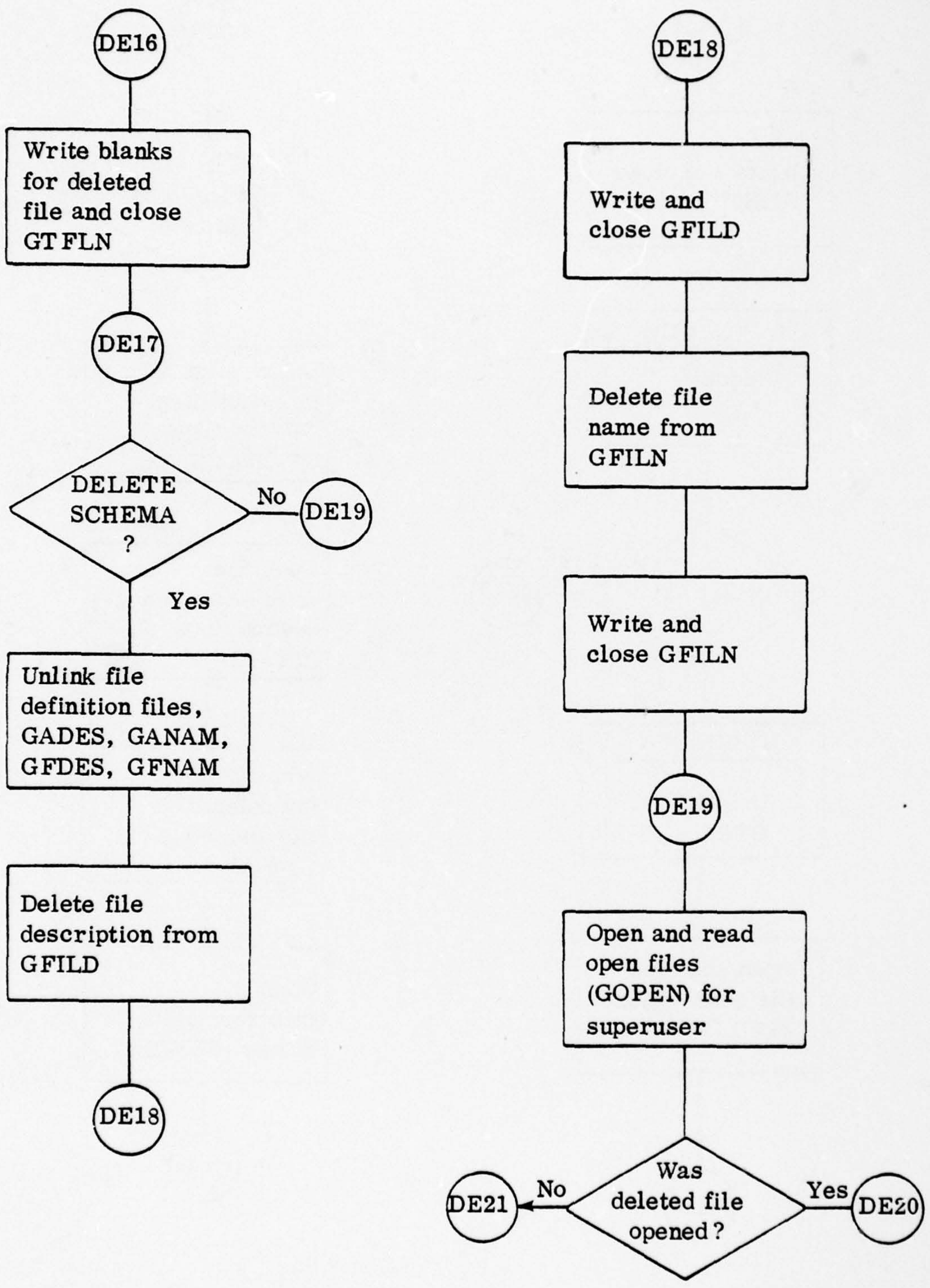

DE16
Write blanks for deleted file and close GTFLN
DE17
DELETE SCHEMA ?
No
DE19
Yes
Unlink file definition files, GADES, GANAM, GFDES, GFNAM
Delete file description from GFILD
DE18
DE18
Write and close GFILD
Delete file name from GFILN
Write and close GFILN
DE19
Open and read open files (GOPEN) for superuser
Was deleted file opened ?
No
DE21
Yes
DE20

Figure 39. DELETE Process Data Flow (Sheet 7 of 12)

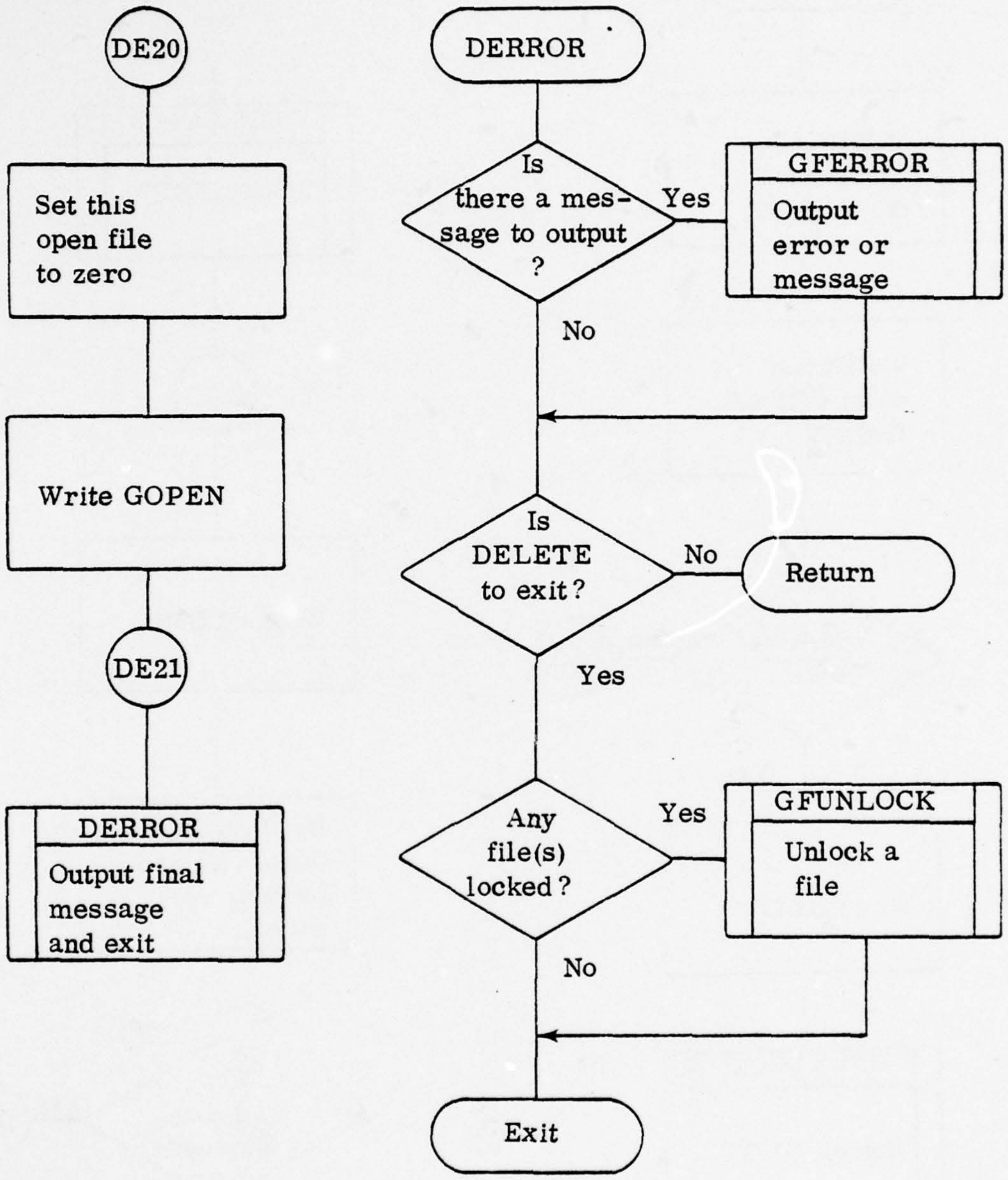


Figure 39. DELETE Process Data Flow (Sheet 8 of 12)

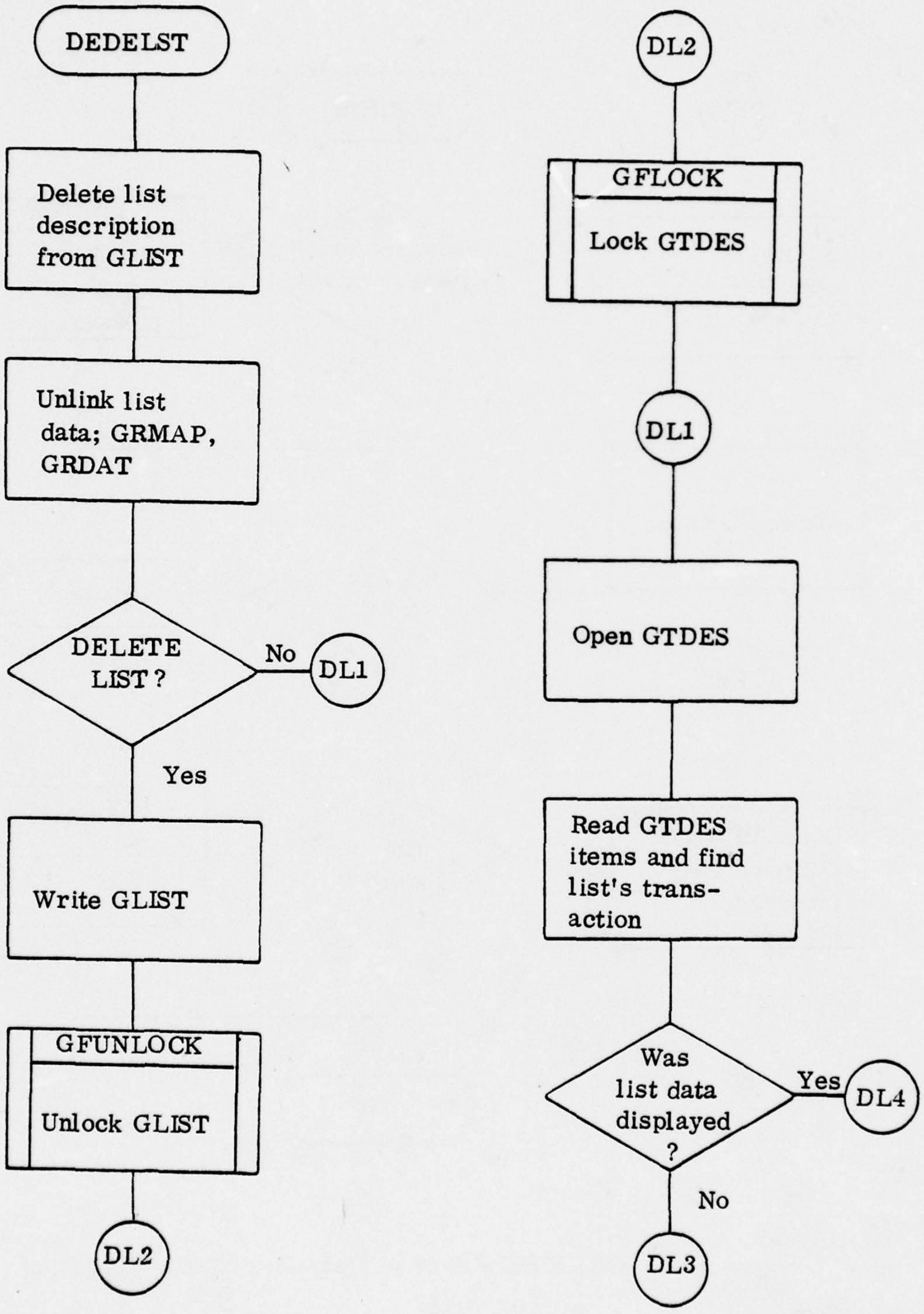

DEDELST
Delete list description from GLIST
Unlink list data; GRMAP, GRDAT
DELETE LIST ?
No
DL1
Yes
Write GLIST
GFUNLOCK
Unlock GLIST
DL2
DL2
GFLOCK
Lock GTDES
DL1
Open GTDES
Read GTDES items and find list's transaction
Was list data displayed ?
Yes
DL4
No
DL3

Figure 39. DELETE Process Data Flow (Sheet 9 of 12)

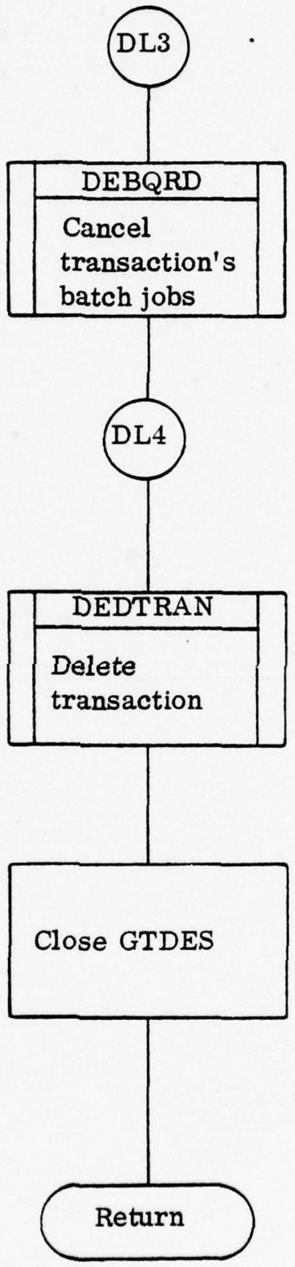


Figure 39. DELETE Process Data Flow (Sheet 10 of 12)

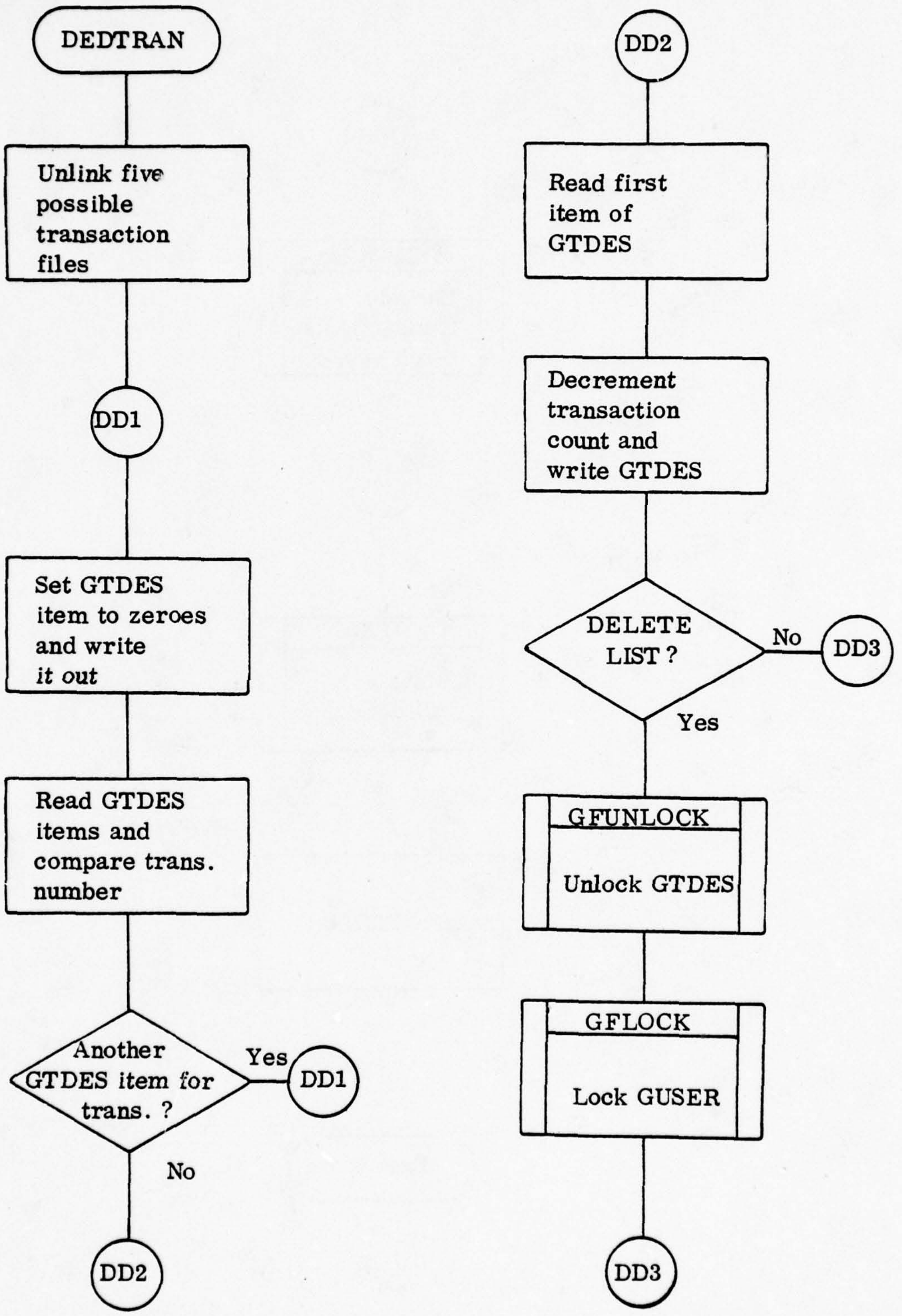

DEDTRAN
Unlink five possible transaction files
DD1
Set GTDES item to zeroes and write it out
Read GTDES items and compare trans. number
Another GTDES item for trans. ?
Yes
DD1
No
DD2
DD2
Read first item of GTDES
Decrement transaction count and write GTDES
DELETE LIST ?
No
DD3
Yes
GFUNLOCK
Unlock GTDES
GFLOCK
Lock GUSER
DD3

Figure 39. DELETE Process Data Flow (Sheet 11 of 12)

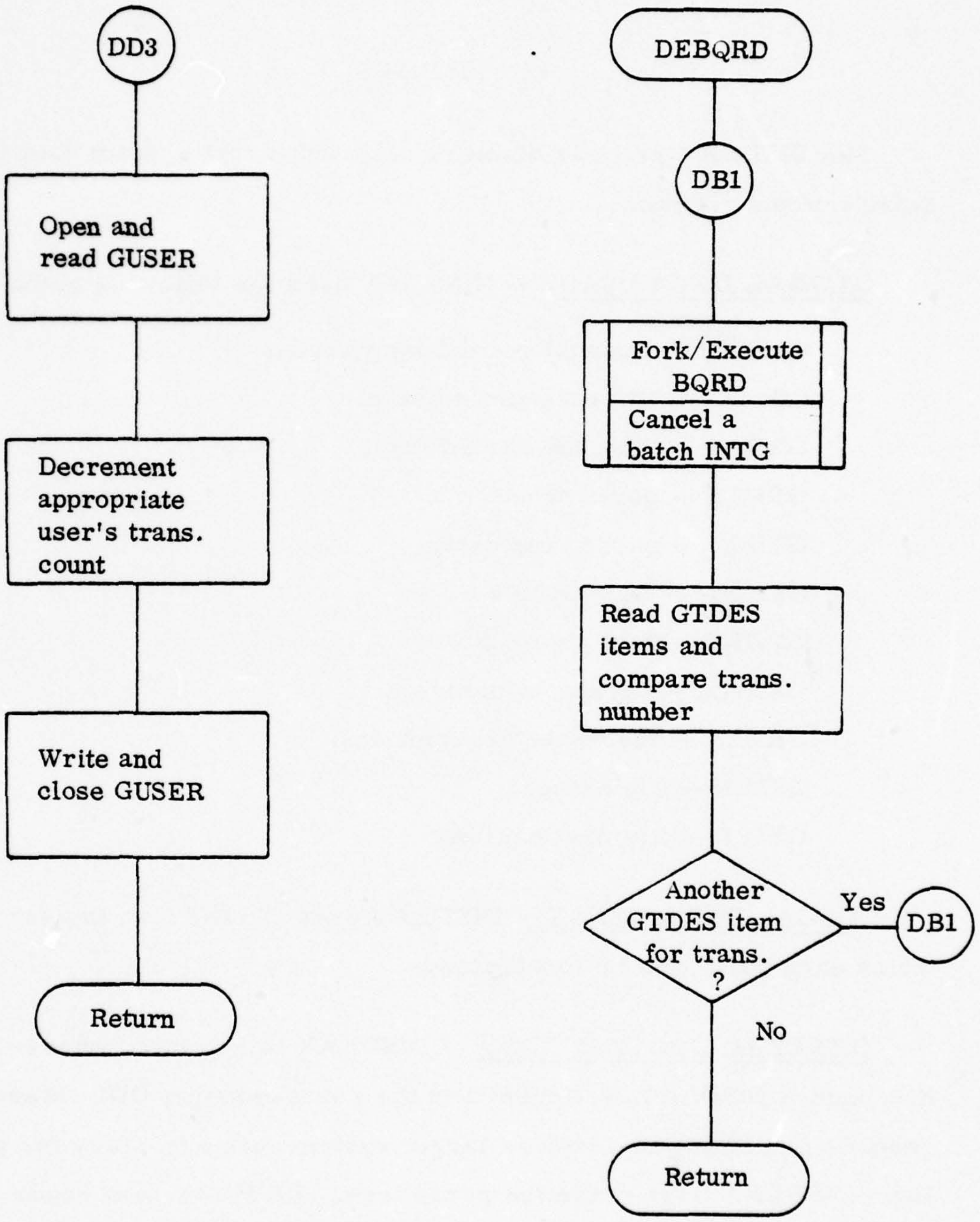

DD3
Open and read GUSER
Decrement appropriate user's trans. count
Write and close GUSER
Return
DEBQRD
DB1
Fork/Execute BQRD
Cancel a batch INTG
Read GTDES items and compare trans. number
Another GTDES item for trans. ?
Yes
DB1
No
Return

Figure 39. DELETE Process Data Flow (Sheet 12 of 12)

## DISPLAY

The DISPLAY process displays selected target system data in a default output format.

GLOBAL DATA USAGE – DISPLAY uses the following global data:

GMPHD – internal record map header.

GRMAP – internal record map.

GRDAT – internal record data.

GTREE – parse tree.

GTDAT – parse tree data.

GFNAM – field names.

GFDES – field descriptions.

GANAM – aggregate names.

GADES – aggregate descriptions.

GFILN – file names.

GFILD – file descriptions.

LOCAL DATA USAGE – DISPLAY uses DILINE, an integer array in which each print line is constructed.

GENERAL PROCESS FLOW – DISPLAY is activated whenever a user has input a DISPLAY statement and the corresponding UDL internal records containing the desired target system data are ready for processing. DISPLAY first reads the parse tree. DISPLAY next reads the file descriptive global data which will be needed to correctly format the data to be displayed. These global data include field names and descriptions

(GFNAM, GFDES), aggregate names and descriptions (GANAM, GADES), and file names and descriptions (GFILN, GFILD).

For each internal record in the list (implicit or explicit) to be processed, DISPLAY utilizes the following logic. A record header is output specifying the record number. If no display list has been specified in the DISPLAY statement, function DIALLOT is called to output all data in the record. If a display list has been specified, each element is processed in the order specified in the list. A character constant is picked up from the parse tree data (GTDAT) and displayed just as it was input. An aggregate name or an aggregate name qualified by the TREE specification is processed by function DITREE. A field name with no subscripts is processed by function DITREE if it is in an aggregate's data set, otherwise it is processed by function DIFDOUT. A field name with subscripts is processed by function DITREE. After all internal records have been displayed, DISPLAY exits and control returns to the calling process. See figure 40 for data flow.

## MAJOR FUNCTION DESCRIPTIONS

DITREE (Display Hierarchical Tree Structure) – DITREE has two input parameters: an aggregate index, and a flag which specifies whether the aggregate, a specified field in the aggregate, or all of the aggregate's tree structure is to be output. For each level of aggregate outside of the input aggregate, that aggregate's name is output. If an aggregate has no occurrences at any point in the tree structure, "NO OCCURRENCES" is output for that aggregate. If a subscripted field is being displayed, only the specified array occurrence is output, and, if that occurrence is missing, the array name, occurrence numer, and "NO OCCURRENCE" are output. Also, for a subscripted field output, the field data are processed by function

DIFDOUT. Otherwise, the input aggregate is processed by function DIALLOT which also receives the input flag value to DITREE. All occur-r-nces of the specified aggregate (or nonsubscripted field) are displayed.

DIALLOT (Display a Data Set) – DIALLOT has three input parameters: an aggregate index; a record map pointer; and a flag which specifies whether all fields and subordinate aggregates for the input aggregate are to be output, all fields and no subordinate aggregates are to be output, or only a specified field is to be output. First, the input aggregate's name is output. If the aggregate has no occurrences, "NO OCCURR-NCES" is displayed. Otherwise, for each occurrence of the aggregate, th aggregate's fields are output by function DIFDOUT and, if subordina aggregates are to be output, each one is processed recursively by function DIALLOT.

DIFDOUT (Output an Aggregate's Fields) – DIFDOUT has three input parameters: an aggregate index, a record map pointer, and a flag which specifies whether all fields or a specified field in the input aggregate are to be displayed. For each field to be displayed, the field's name and an equals sign are output. If the field contains data, those data are output either as a single value or as multiple values if the field is multivalued.

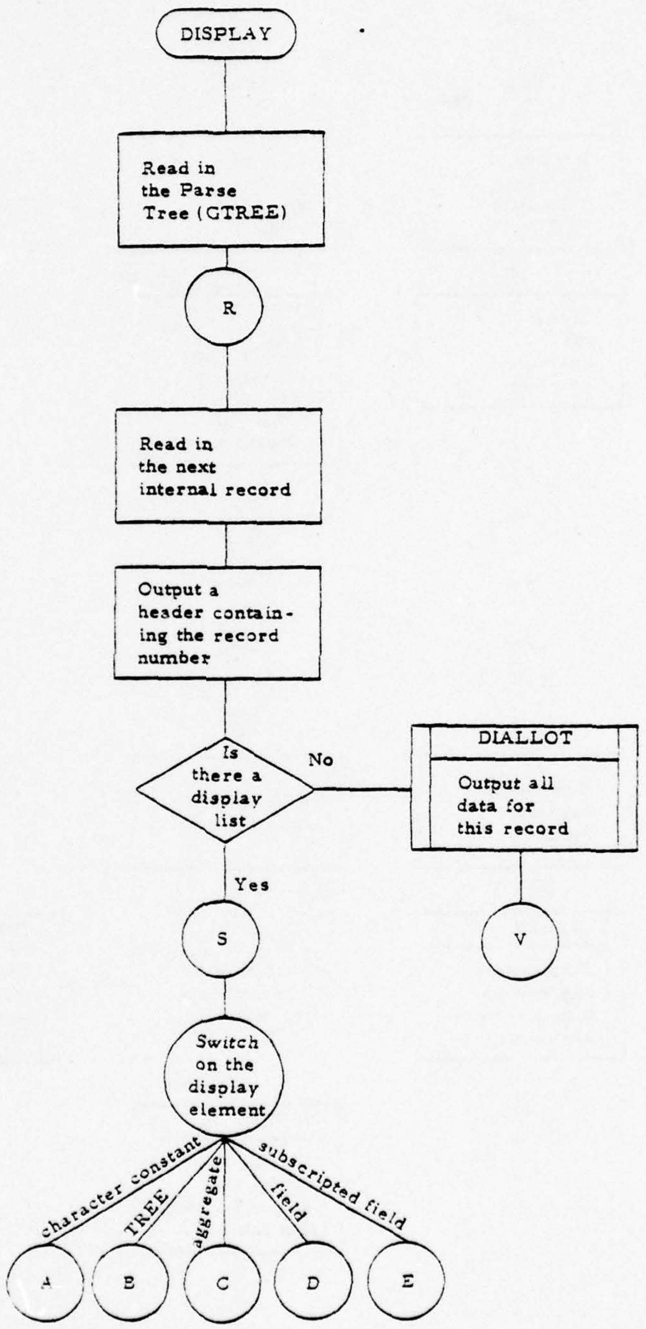

DISPLAY
Read in the Parse Tree (GTREE)
R
Read in the next internal record
Output a header contain-ing the record number
Is there a display list
No
DIALLOT
Output all data for this record
Yes
S
V
Switch on the display element
character constant
TREE
aggregate
field
subscripted field
A
B
C
D
E

Figure 40. DISPLAY Process Data Flow (Sheet 1 of 8)

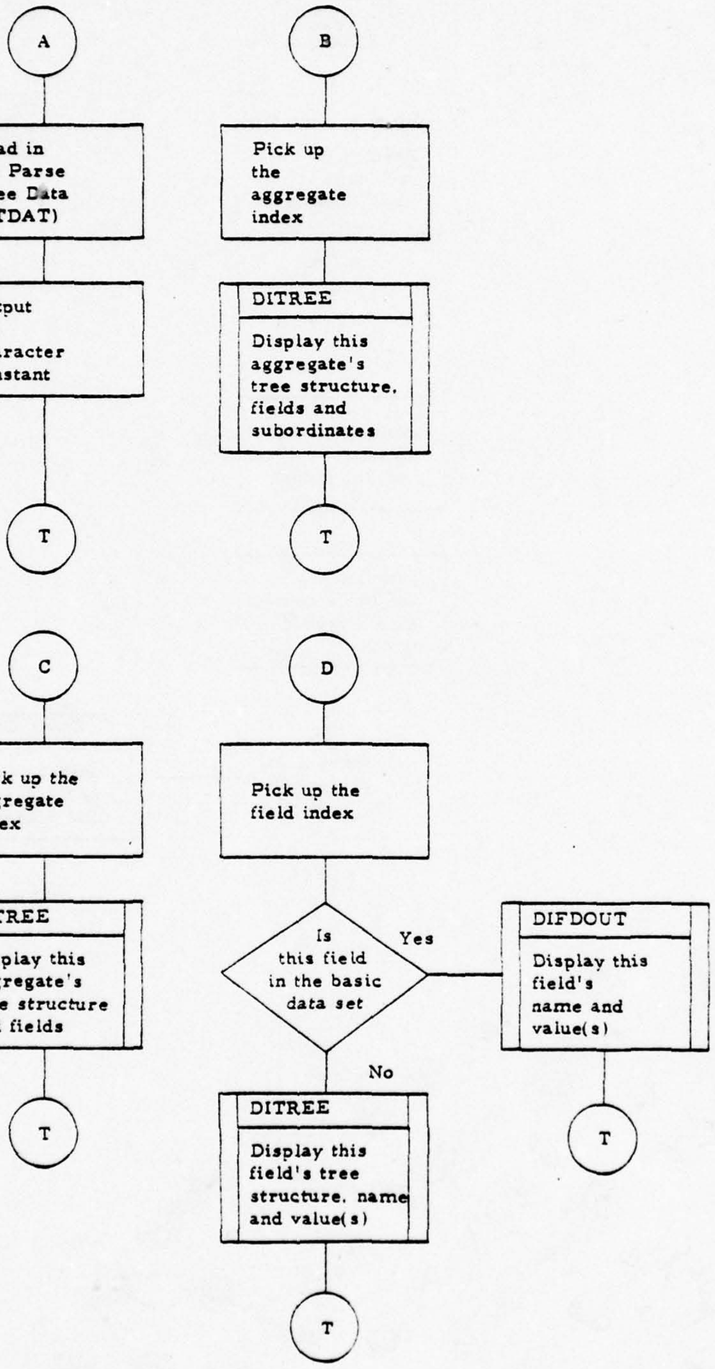

A
Read in the Parse Tree Data (GTDAT)
Output the character constant
T
B
Pick up the aggregate index
DITREE
Display this aggregate's tree structure, fields and subordinates
T
C
Pick up the aggregate index
DITREE
Display this aggregate's tree structure and fields
T
D
Pick up the field index
Is this field in the basic data set
Yes
DIFDOUT
Display this field's name and value(s)
T
No
DITREE
Display this field's tree structure, name and value(s)
T

Figure 40. DISPLAY Process Data Flow (Sheet 2 of 8)

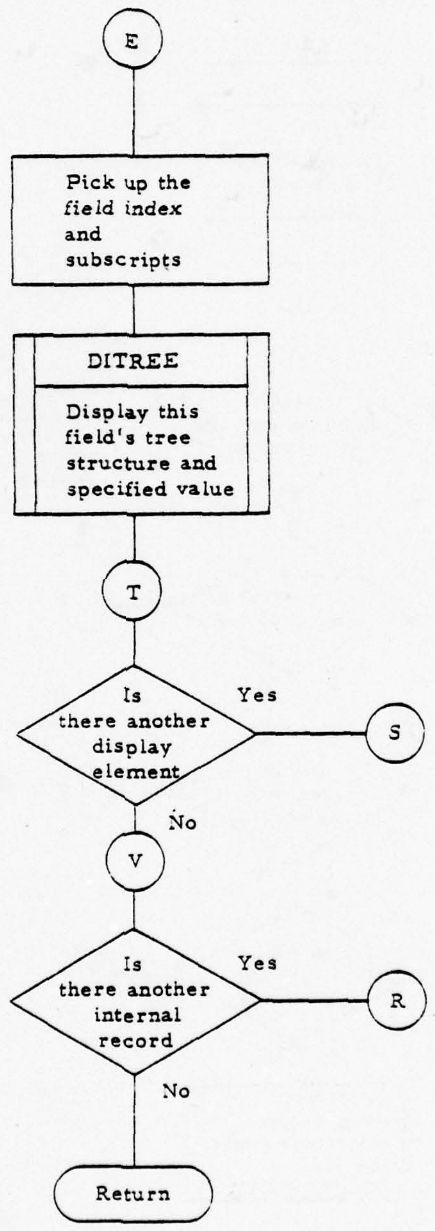

E
Pick up the *field index* and subscripts
DITREE
Display this field's tree structure and specified value
T
Is there another display element
Yes
S
No
V
Is there another internal record
Yes
R
No
Return

Figure 40. DISPLAY Process Data Flow (Sheet 3 of 8)

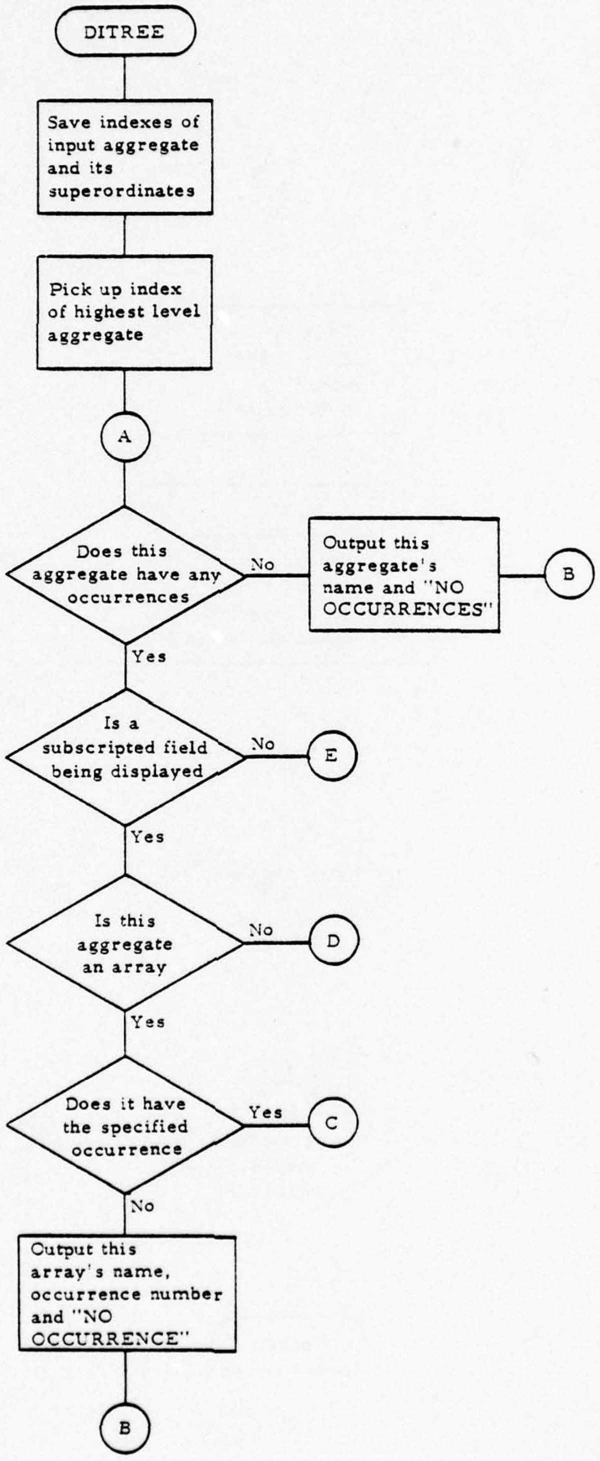

DITREE
Save indexes of input aggregate and its superordinates
Pick up index of highest level aggregate
A
Does this aggregate have any occurrences
No
Output this aggregate's name and "NO OCCURRENCES"
B
Yes
Is a subscripted field being displayed
No
E
Yes
Is this aggregate an array
No
D
Yes
Does it have the specified occurrence
Yes
C
No
Output this array's name, occurrence number and "NO OCCURRENCE"
B

Figure 40. DISPLAY Process Data Flow (Sheet 4 of 8)

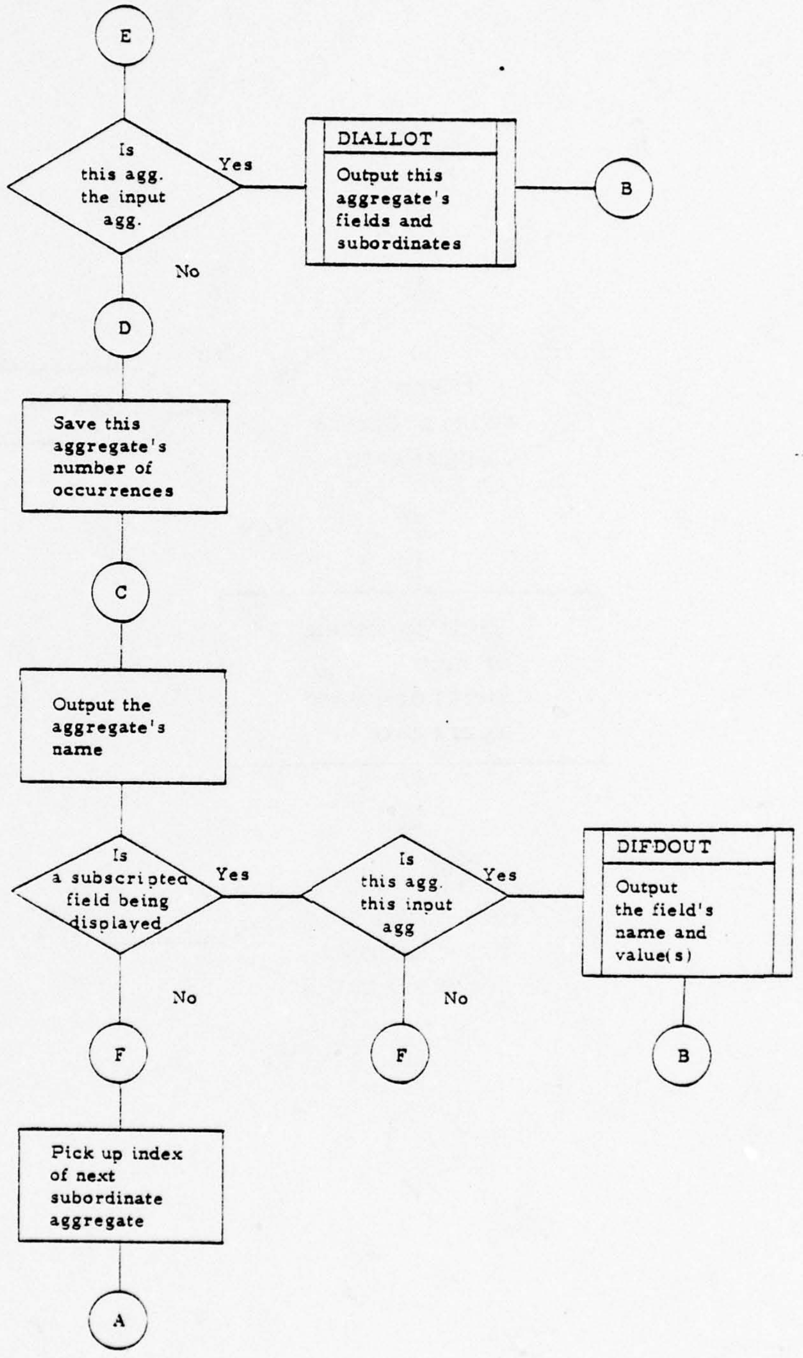

E
Is this agg. the input agg.
Yes
No
DIALLOT
Output this aggregate's fields and subordinates
B
D
Save this aggregate's number of occurrences
C
Output the aggregate's name
Is a subscripted field being displayed
Yes
No
Is this agg. this input agg
Yes
No
DIFDOUT
Output the field's name and value(s)
F
F
B
Pick up index of next subordinate aggregate
A

Figure 40. DISPLAY Process Data Flow (Sheet 5 of 8)

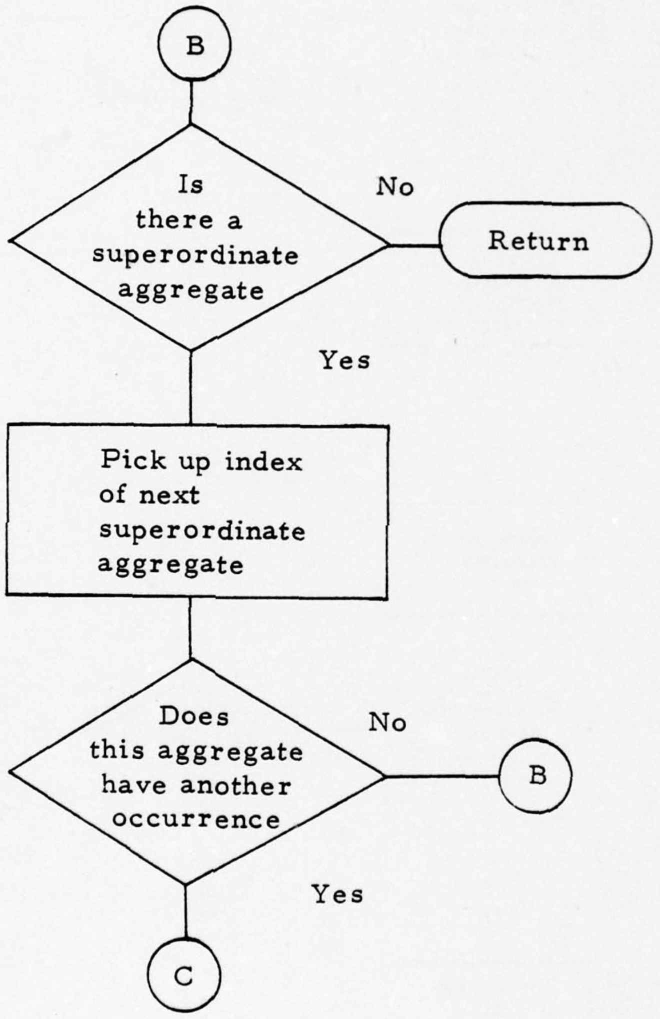

B
Is there a superordinate aggregate
No
Return
Yes
Pick up index of next superordinate aggregate
Does this aggregate have another occurrence
No
B
Yes
C

Figure 40. DISPLAY Process Data Flow (Sheet 6 of 8)

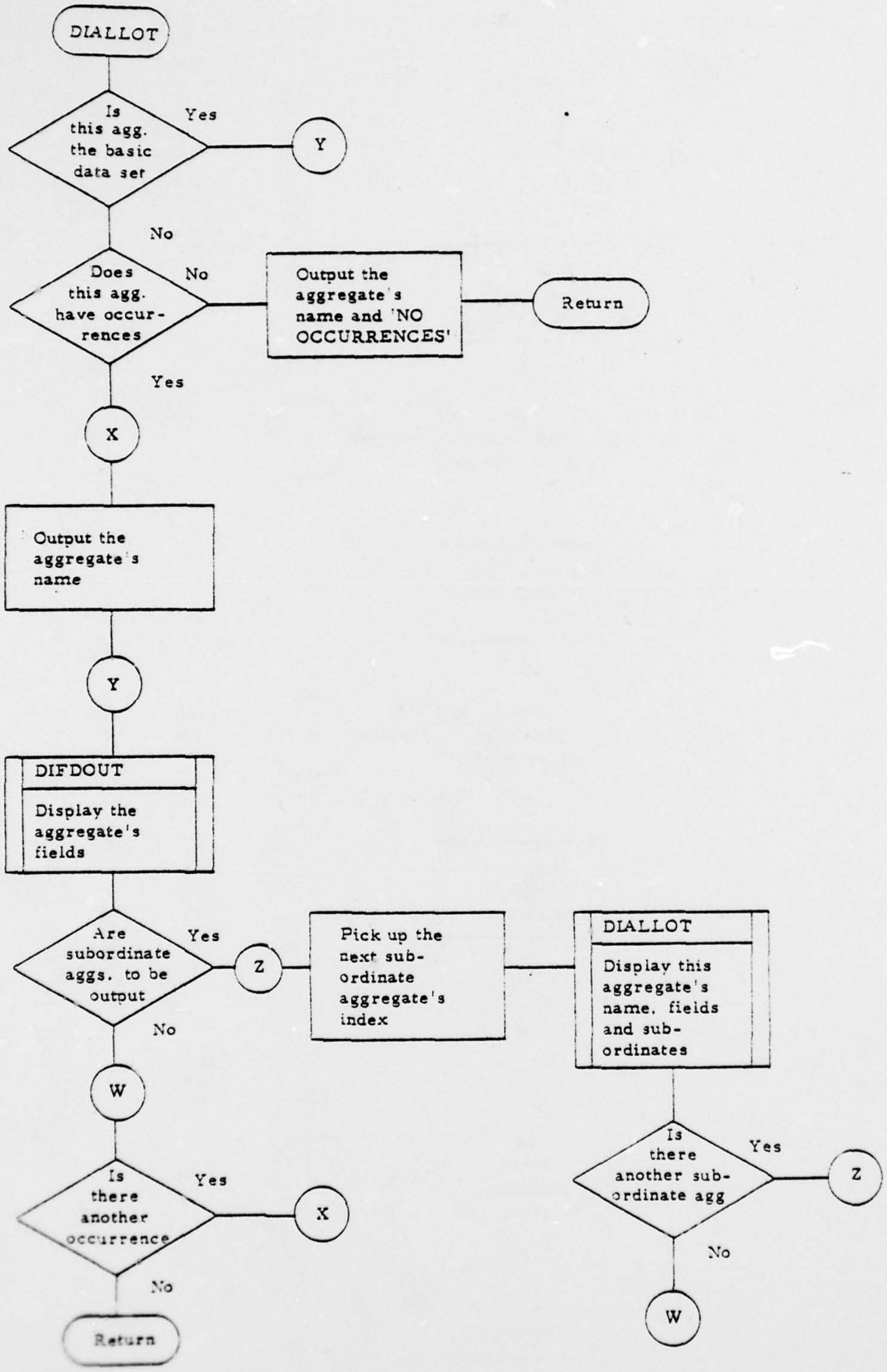

DIALLOT
Is this agg. the basic data set
Yes
Y
No
Does this agg. have occurrences
No
Output the aggregate's name and 'NO OCCURRENCES'
Return
Yes
X
Output the aggregate's name
Y
DIFDOUT
Display the aggregate's fields
Are subordinate aggs. to be output
Yes
Z
Pick up the next subordinate aggregate's index
DIALLOT
Display this aggregate's name, fields and subordinates
No
W
Is there another occurrence
Yes
X
No
Return
Is there another subordinate agg
Yes
Z
No
W

Figure 40. DISPLAY Process Data Flow (Sheet 7 of 8)

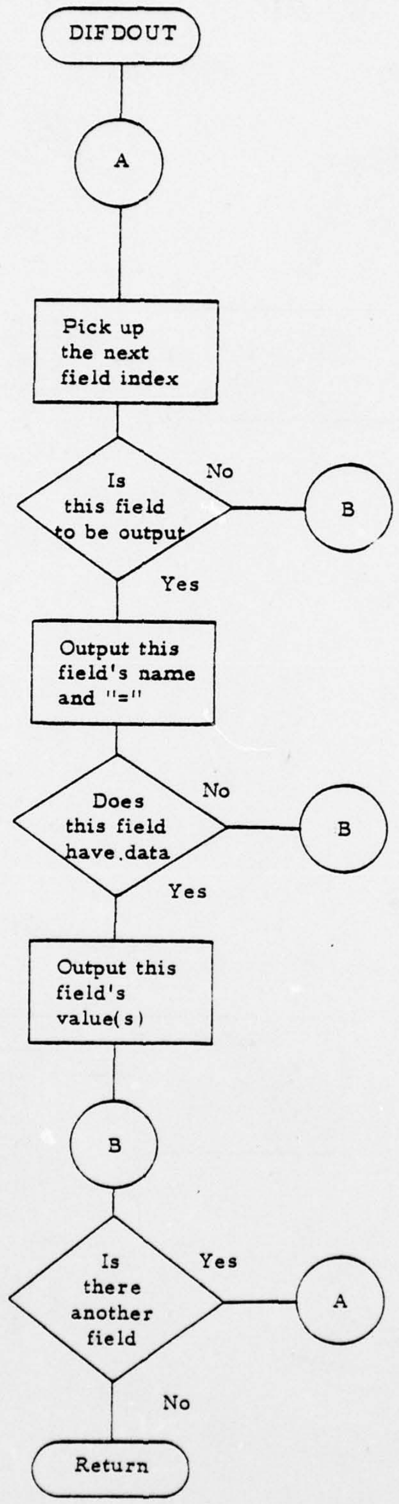

DIFDOUT
A
Pick up the next field index
Is this field to be output
No
B
Yes
Output this field's name and "="
Does this field have data
No
B
Yes
Output this field's value(s)
B
Is there another field
Yes
A
No
Return

Figure 40. DISPLAY Process Data Flow (Sheet 8 of 8)

## EXEC (ADAPT Executive)

The EXEC process initializes and controls the ADAPT environment for users while they are logged on to ADAPT. As each user types in an ADAPT statement or command, EXEC parses this input and builds a parse tree. EXEC then has the statement validated, if applicable, and either processes the statement or calls the appropriate processor.

GLOBAL DATA USAGE – EXEC uses the following global data:

GTREE – parse tree.

GTDAT – parse tree data.

LOCAL DATA USAGE – EXEC uses the following local data:

LXKYWRD – keywords.

LXTOKEN – keyword token types.

GENERAL PROCESS FLOW – EXEC is activated whenever a user successfully logs onto ADAPT and remains in control until the user quits. EXEC calls the LEX/parse initialization (LXINIT) which causes the next input statement to be read in and parsed, and a parse tree to be built. If a syntax error is encountered in the statement, an error message is output and EXEC goes on to the next statement. Otherwise, the statement's primary token type is picked up and EXEC determines whether the statement needs semantical validation. If so, the VALIDATE process is called to validate and modify the statement parse tree. VALIDATE will call the appropriate process if there are no semantic errors. Otherwise, EXEC determines which process is to receive the statement for processing and calls that process. EXEC itself processes the EXECUTE and MODE commands. If the statement is a MODE command, EXEC sets or clears a flag which is used to determine whether the statement is to be processed or only validated. If the statement is EXECUTE, the user-specified input file is opened. If ECHO has been specified on the EXECUTE command, a flag is set to specify that all statements from the user's file are to be echoed to the terminal. If the EXECUTE command specifies SCHEMA, the DDL process is called and is passed the echo flag and the input file specification. When EOF is reached on a user specified input file, the input file specification is reset to the terminal and the echo flag is cleared. If an internal system error occurs in a process called by EXEC or in EXEC itself, an error message is output and ADAPT is terminated for that user. After a QUIT command has been processed, EXEC also terminates for the user. See figure 41 for data flow.

## MAJOR FUNCTION DESCRIPTIONS

LXINIT (LEX/Parse Initialization) – LXINIT is merely an initialization function for the LEX/parse. It initializes pertinent data including a flag which it returns to the calling function to indicate EOF, syntax error, or statement accepted. LXINIT calls the parsing function (YYPARSE) and returns with the flag value.

YYPARSE (Parser) – YYPARSE is a parser which is produced by the YACC program running under UNIX. YACC produces the parser as well as a set of tables which the parser uses to organize the tokens passed to it by the lexical analyzer (YYLEX). These tables reflect the grammer of the various UDL statements. YYPARSE calls YYLEX, which returns a value called a token type. If the token type is invalid according to the input statement syntax rules, an error message is output and YYPARSE calls YYLEX continuously until an end-of-input token type is returned. YYPARSE then returns with the return flag set to reflect a syntax error. If the token type is valid according to the input rules, the action specified in the YACC run is performed. The actions that can be invoked are: no action, build a parse tree node with the specified number of legs, store the final token type and write out the parse tree and parse tree data, start a list, add an element to the current list, and store the list in the parse tree. After the action is performed, YYPARSE calls YYLEX for the next token type. When the end-of-input token type is received and its action performed, YYPARSE returns with the return flag set to reflect statement accepted. If YYLEX encounters an EOF, it returns an end-of-input token type to YYPARSE and sets the return flag to reflect EOF.

YYLEX (Lexical Analyzer) – YYLEX is the function which actually reads and processes each UDL statement. YYLEX reads and saves each input character until a delimiter is encountered. The delimiters for ADAPT I are: comma, newline (carriage return), single quote mark, percent sign, space, left parenthesis, right parenthesis, and semi-colon. If an EOF is encountered, the return flag is set to reflect EOF and an end-of-input token type is returned. If a delimiter is encountered by itself, it is either passed to the parser (comma, left or right parenthesis, or semi-colon), it is ignored, or it causes YYLEX to change its input mode; e. g. , single quote mark causes YYLEX to read characters until another single quote mark is input. When a delimiter terminates a string of characters, YYLEX must determine what the characters represent. If the characters form a UDL keyword, YYLEX returns with that keyword's token type. Otherwise, YYLEX determines the type of constant which is formed by the characters (name, integer, numeric constant, geographic constant, or character constant), stores the value of the constant in the parse tree data (GTDAT), stores a terminal node in the parse tree (GTREE), and returns with the constant's token type. If YYLEX encounters an error (i. e. , too many characters input), it returns an invalid token type to the parser and thereby generates a syntax error message.

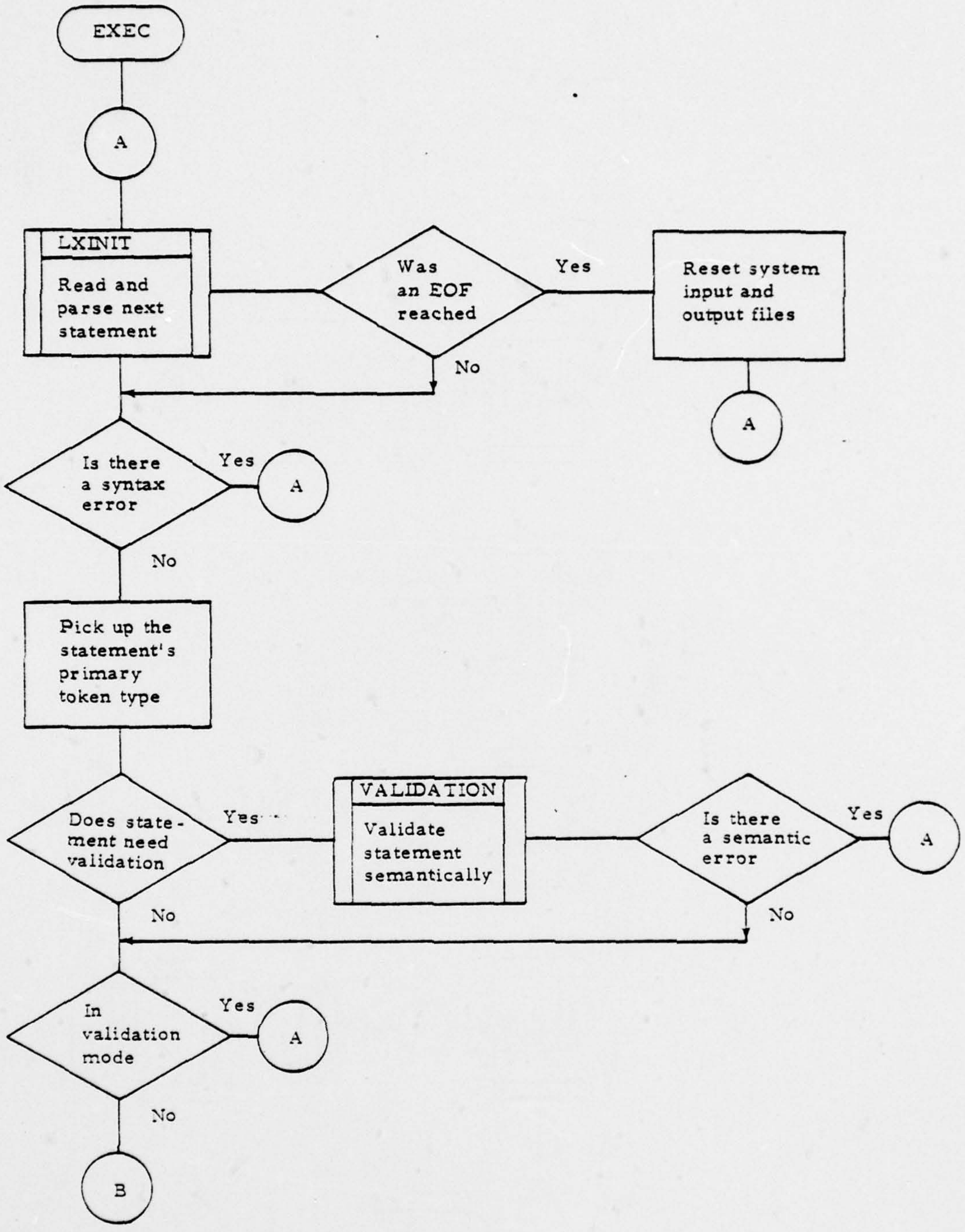

EXEC
A
LXINIT
Read and parse next statement
Was an EOF reached
Yes
No
Reset system input and output files
A
Is there a syntax error
Yes
A
No
Pick up the statement's primary token type
Does statement need validation
Yes
VALIDATION
Validate statement semantically
Is there a semantic error
Yes
A
No
No
In validation mode
Yes
A
No
B

Figure 41. EXEC Process Data Flow (Sheet 1 of 6)

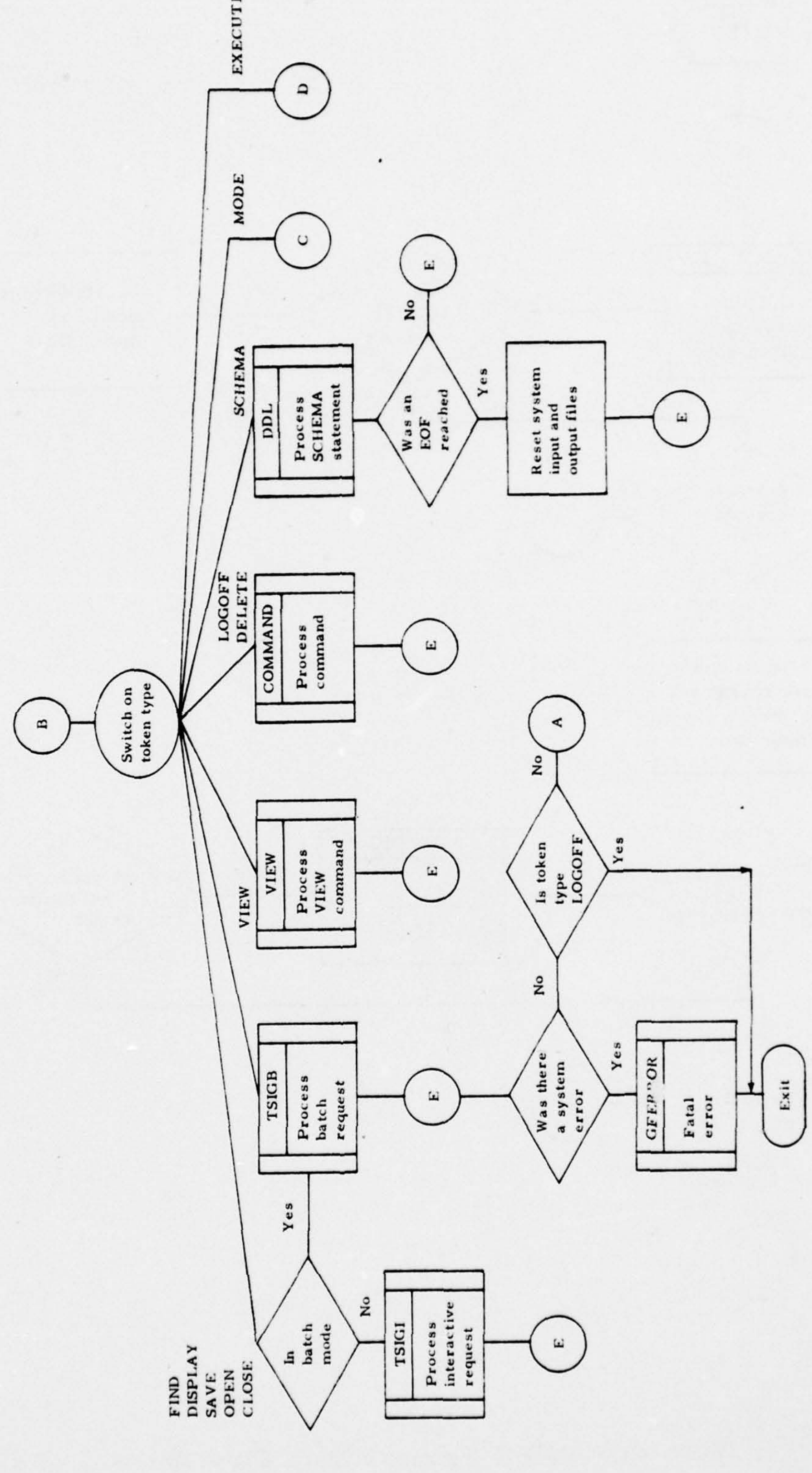


Figure 41. EXEC Process Data Flow (Sheet 2 of 6)

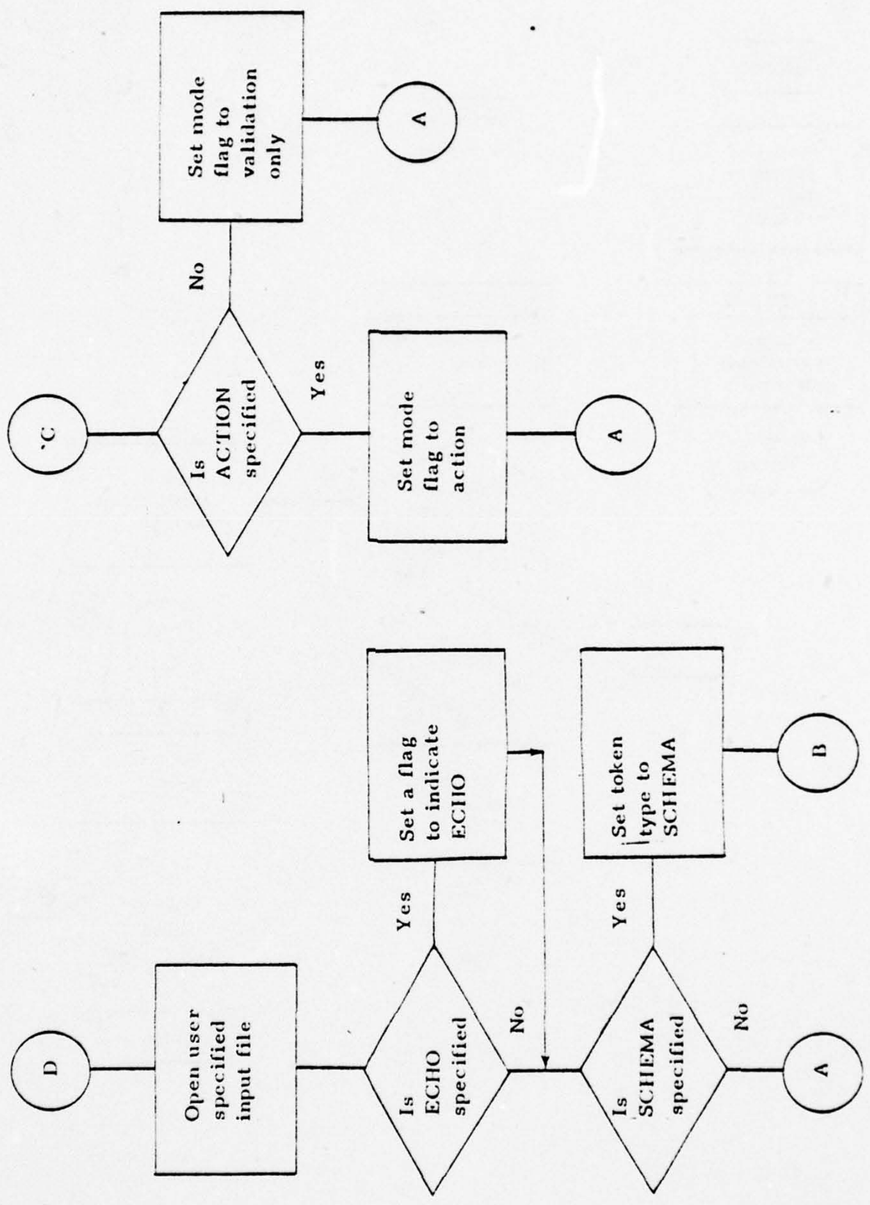


Figure 41. EXEC Process Data Flow (Sheet 3 of 6)

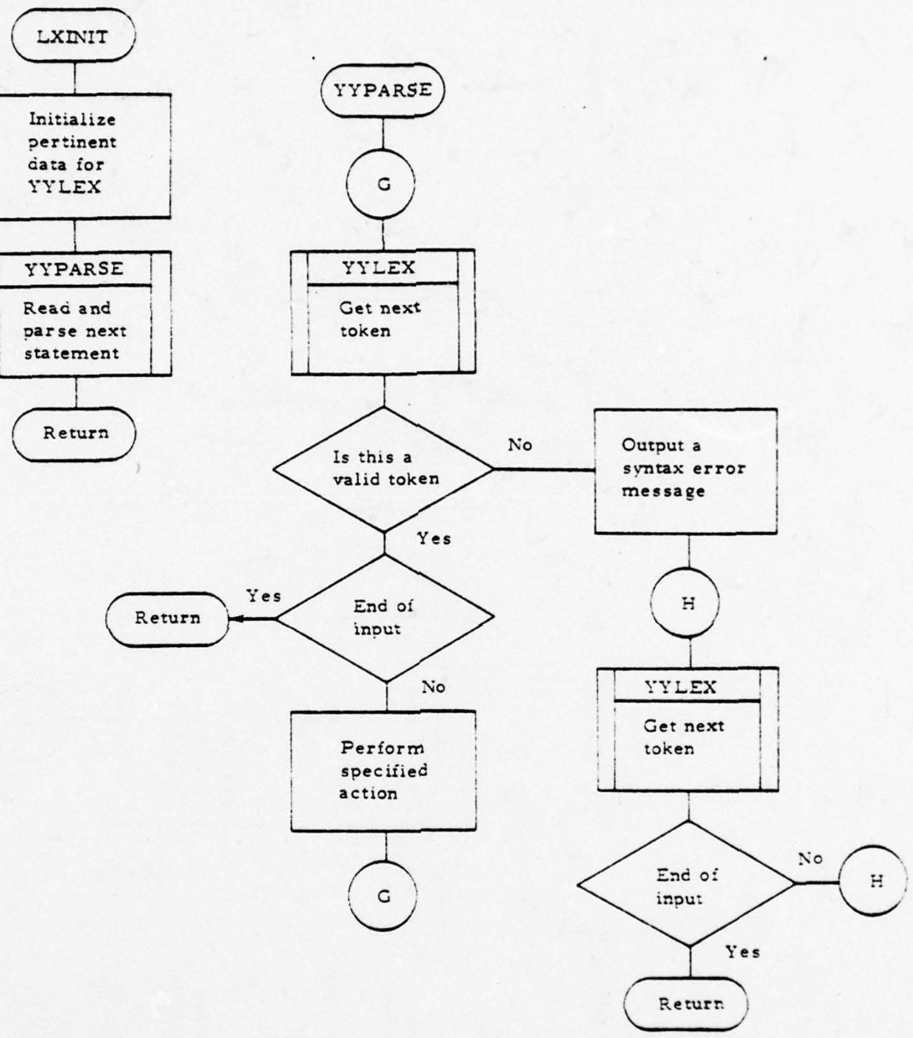


Figure 41. EXEC Process Data Flow (Sheet 4 of 6)

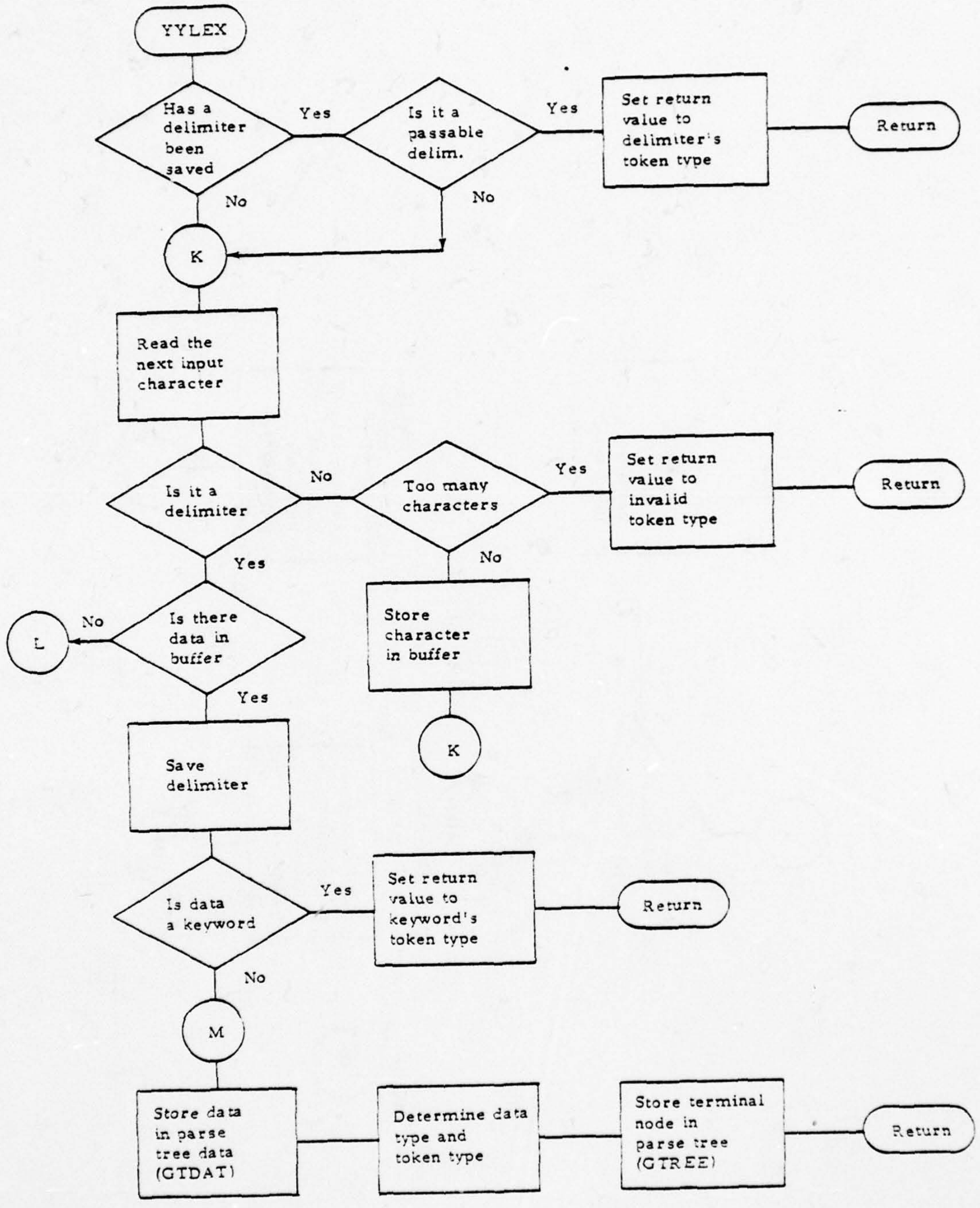

YYLEX
Has a delimiter been saved
Yes
No
Is it a passable delim.
Yes
No
Set return value to delimiter's token type
Return
K
Read the next input character
Is it a delimiter
No
Yes
Too many characters
Yes
No
Set return value to invalid token type
Return
Store character in buffer
K
Is there data in buffer
No
Yes
L
Save delimiter
Is data a keyword
Yes
No
Set return value to keyword's token type
Return
M
Store data in parse tree data (GTDAT)
Determine data type and token type
Store terminal node in parse tree (GTREE)
Return

Figure 41. EXEC Process Data Flow (Sheet 5 of 6)

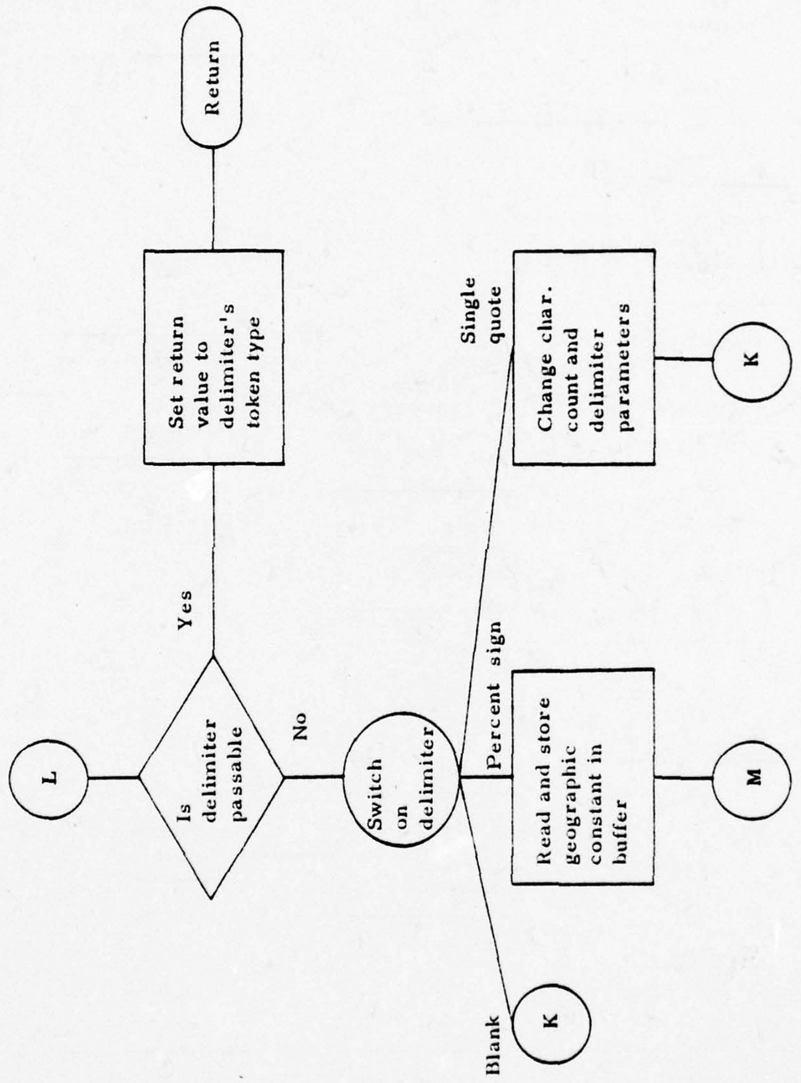

L
Is delimiter passable
Yes
Set return value to delimiter's token type
Return
No
Switch on delimiter
Percent sign
Read and store geographic constant in buffer
M
Single quote
Change char. count and delimiter parameters
K
Blank
K

Figure 41. EXEC Process Data Flow (Sheet 6 of 6)

## NETRES (Network Response Processor)

The NETRES process constitutes the other half of ADAPT I's batch query processing. Where TSIGB is responsible for initiating batch transactions, NETRES is responsible for processing the subsequent transaction responses. NETRES' responsibilities include the processing of batch responses, maintenance of the transaction descriptions file (GTDES), and the removal of various temporary record data and implicit list names that might exist as a result of the transaction.

GLOBAL DATA USAGE – NETRES utilizes the following global data:

| | |
|---|---|
| GTDES | – transaction descriptions. |
| GTREE | – parse tree. |
| GTDAT | – parse tree data. |
| GRMAP | – internal record map. |
| GRDAT | – internal record data. |
| GLIST | – list names and descriptions. |

LOCAL DATA USAGE – NETRES (specifically, NETRESD) uses the following local data:

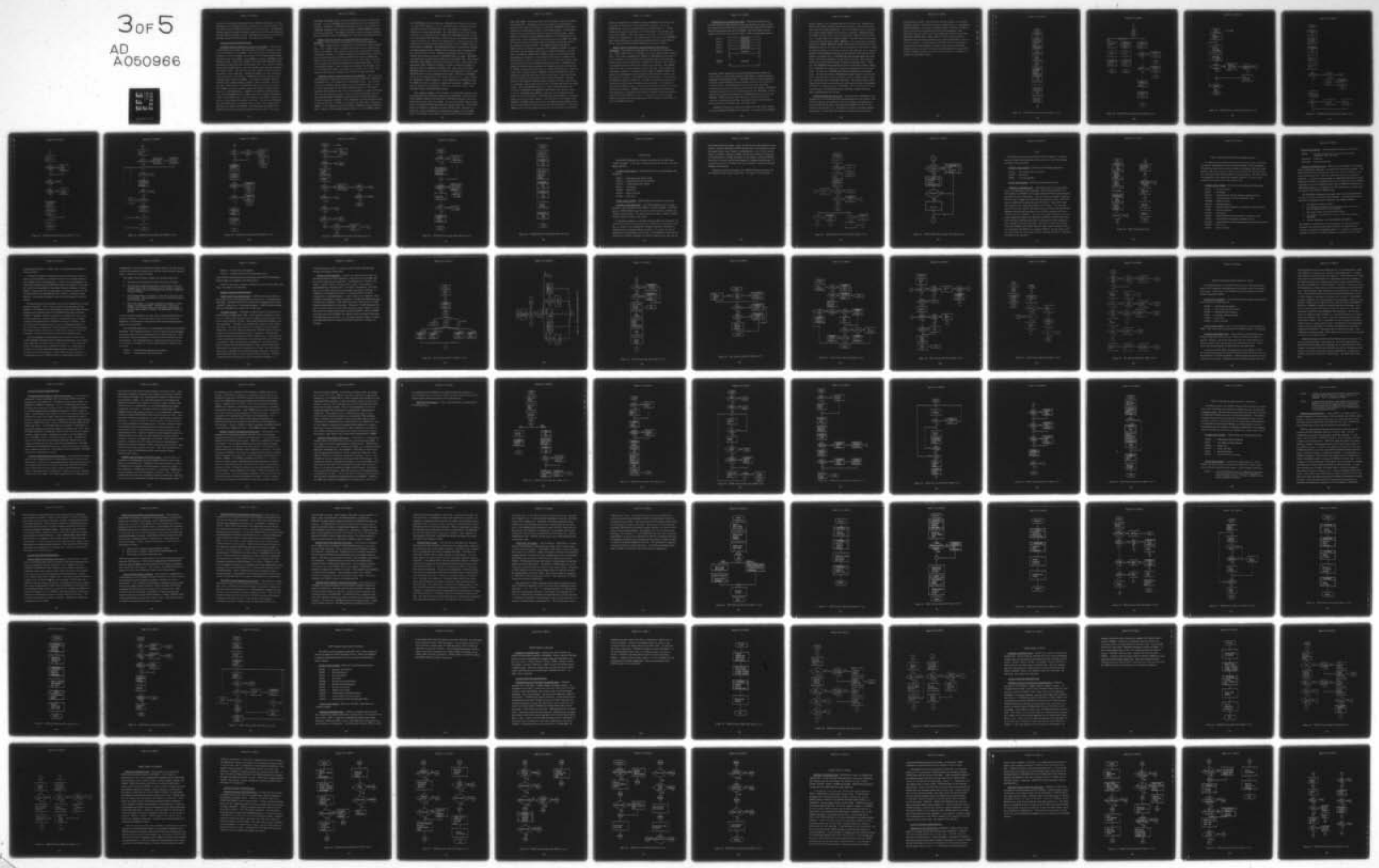
NERC – Record-id Control. NETRESD uses this data structure to contain the hash table plus related information used in controlling unique record-ids for multi-INTG sequences.

GENERAL PROCESS FLOW – Under ADAPT I the NETRES process is actually composed of two processes, NETRESR for RYETIP and NETRESD for DIAOLS. However, since both hosts are batch, the upper level data flow of both processes is quite similar and is described here as a unit. However, differences do occur with respect to the multi-INTG sequences that can occur with DIAOLS transformations. These differences are described below as separate function descriptions.

NETRES is activated by the Batch Query and Response Dispatcher (BQRD) process of TAS when a response comes back from some batch host resulting from an INTG (interrogation) sent by ADAPT I. The host to which the query was sent will determine which NETRES process is called (i.e., TSIG1 sends batch queries to RYETIP and specifies NETRESR as the response processor, and similarly, TSIG2 queries DIAOLS and specifies NETRESD as its response processor). BQRD passes to NETRES the response type (e.g., ANSR, ABRT(FAULT)), the TAS assigned JOBID, the pathname of the response file, and the classification of the response. Generally, both NETRES processes then read in the response and generate normalized response lines, removing host generated classification headers that may be present. It is important that only anticipated output from the host be passed to the TSOT process. The normalized output is written out to cLTN. Also an important responsibility of the NETRES process is the processing and maintenance of the GTDES data structure. For each job submitted to a batch host, there exists an item in GTDES describing this job (or logical transaction, as it is called in ADAPT I). Information such as the TAS assigned JOBID, ADAPT assigned Logical Transaction Number (LTN), file ID, associated list ID, is contained in GTDES. GTDES items are controlled by the mode field, GTMOD, which is set initially to active by the appropriate TSIGB process and is set to completed, aborted, or lost by the appropriate NETRES process. The VIEW process is responsible for actually removing the GTDES item after the user has perused the response. Following the building of the logical response output in cLTN, NETRES calls the appropriate TSOT process for the generation of GRMAP/GRDAT record data representing the host response. Upon completion of this, NETRES then checks

AD-A050 966 LOGICON INC SAN DIEGO CALIF F/G 5/2
ADAPT I FINAL FUNCTIONAL AND SYSTEM DESIGN SPECIFICATION.(U)
JAN 78 L R ERICKSON, M E SOLEGLAD N00014-76-C-0899
UNCLASSIFIED 76-C-0899-2-F NL

3 OF 5
AD
A050966



to see if the response was generated from a DISPLAY statement. If it was, NETRES calls the DISPLAY process, passing to it the file descriptors of the saved parse trees, GTREE and GTDAT, in aLTN and bLTN. The DISPLAY process outputs the display information onto the user's output file associated with this logical transaction, dLTN. NETRES then removes all LTN associated files except for dLTN and exits. See figure 42 for data flow.

## MAJOR FUNCTION DESCRIPTIONS

NETRESR (Network Response Processor for RYETIP) — This function is responsible for all upper level processing of network responses coming from the RYETIP host (via BQRD). This function locks the transaction descriptions structure, GTDES, and searches it for the TAS assigned JOBID received as input from BQRD. If the JOBID is not found, NETRESR unlocks GTDES and exits. Otherwise, GTDES is unlocked and the appropriate file pathnames are generated from the associated LTN. Next, dLTN is created and external variable FOUT is set to its file descriptor thereby redirecting the output to dLTN. Then the user's list file, GLIST, is read in for later processing. Record data files, GRMAP and GRDAT, are then created for TSOT processing of the response. NETRESR processing switches on the response type received from BQRD. Five response types are possible: ABRT(FAULT), ABRT(CAUSE), LOGOUT, LOST, and ANSR. The first two types, ABRT(FAULT) and ABRT(CAUSE), are processed as follows: The GTDES item is set to an abort status, NECLNUP is called for final clean up, and NETRESR exits. LOGOUT and LOST are handled together also, where NETRESR sets the GTDES item to a lost status, NECLNUP is called, and NETRESR exits. The last response type, ANSR, implies that the query was satisfied successfully by the host. For this case NETRESR sets the GTDES item to a completed status and then calls the appropriate TSOT process (TSOT1 for RYETIP). TSOT generates the GRMAP and GRDAT data from the normalized response found in cLTN. Upon return from TSOT, NETRESR checks to see if the TSOT process returned an error condition. If it did, NETRESR calls NECLNUP with an error condition and then exits. Otherwise, NETRESR determines if the query was initiated via a DISPLAY statement and, if it was, proceeds to call the DISPLAY process for further processing. The DISPLAY process outputs the specified display data to dLTN and returns to NETRESR, which then calls NECLNUP and exits.

NEPOL (Physical Response to Logical Response Processor for NETRESR) — This function is responsible for converting host responses (i.e., ANSR, ABRT FAULT or CAUSE) into a set of uniform length response lines. First, NEPOL reads in a logical line of data from the response file (passed from BQRD). Each character of this line is searched for a newline or new page character. If a match is made then the saved portion of the line (each scanned character is moved into another line buffer) is padded with blanks to ensure that each host response line is the same length. Then this line is scanned for a potential host generated classification header. If it is a classification header, it and the blank line following it are dropped. If it isn't a classification header, the logical line is written out to cLTN. This continues until the entire response from the host has been processed.

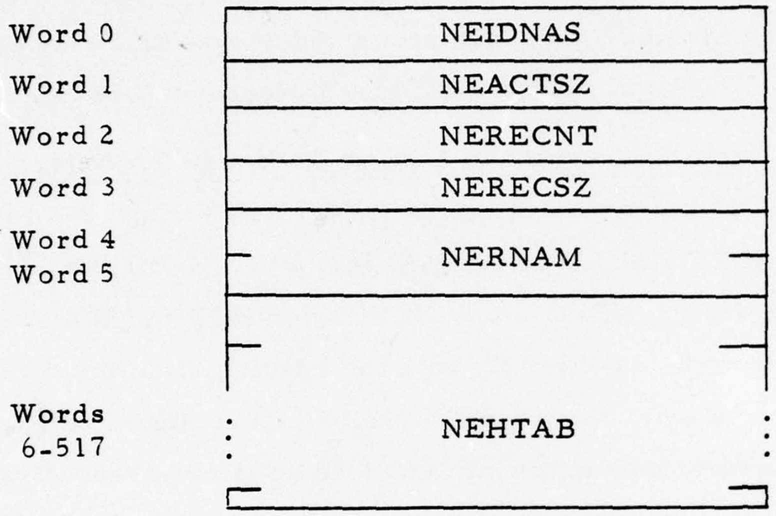
*NETRESD (Network Response Processor for DIAOLS)* — This function is responsible for all upper level processing of network responses coming from the DIAOLS host. It is activated by the TAS BQRD process. It is possible under UDL-to-DIAOLS transformations to have a situation where, in order to satisfy semantically the UDL FIND statement sequence (in this case a dependent FIND statement with one or more geographic or scoped rel-terms), more than one INTG is necessary. This is called a multi-INTG sequence. This fact alone makes NETRESD considerably more complicated than its RYETIP counterpart, NETESR. First, NETRESD must lock GTDES and find the appropriate item in GTDES by scanning for the TAS JOBID (as passed by BQRD). If this JOBID is not found in GTDES, NETRESD unlocks GTDES and exits. However, if the normal situation occurs (i.e., the JOBID is found) then NETRESD continues processing. NETRESD first checks to see if this is a single-INTG sequence. If it is, NETRESD checks further for an abort condition (a response classification other than ANSR) and updates this GTDES item's status appropriately, calls NECLNUP for final clean up and then exits. If the response type is ANSR, then NETRESD proceeds to read in the appropriate TDL data files in order to extract the element-short-name of the record-id as it is known by the DIAOLS host. At this time it also calculates the size of the record-id field and places it into the NERC structure (see writeup on NEHASH). Now NETRESD proceeds to process the response by making repeated calls to NEPOL which produces logical response lines and writes them out to cLTN. NEPOL also searches for the record-id element-short-name and returns to NETRESD when it is found. NETRESD increments the record hit count in NERC for each NEPOL call. This process terminates when NEPOL returns with a status of no data. At this time, the appropriate TSOT (TSOT2 for DIAOLS) is called to convert the logical response in cLTN into GRMAP and GRDAT data. Upon return from TSOT, NETRESD checks to see if an error status was returned and, if it was, calls NECLNUP with an error condition and then exits. Otherwise, NETRESD checks on the statement type of the logical transaction (i. e. , was it initiated by a DISPLAY or SAVE statement). If the statement was a DISPLAY, then the DISPLAY process is called to complete the transaction. DISPLAY proceeds to output the requested data to the user's output file, dLTN. Then NETRESD finally calls NECLNUP and exits.

If the response happens to be an instance of a multi-INTG sequence, the following additional processing is performed by NETRESD. Prior to entering the NEPOL sequence described above, NETRESD must also read in the multi-INTG control item from GTDES. For each multi-INTG sequence there exists a control item which represents the logical transaction. Actual INTGs sent out by TSIG2 are represented by other GTDES items, referred to as instances, and are identified by a subtransaction number in their GTDES items. Accordingly, each instance (a separate INTG) must be assigned a unique TAS JOBID. The control item, plus all instance items, are related by the same LTN. Depending on the current state of the multi-INTG sequence and the state of the instance just received, NETRESD goes through somewhat convoluted logic in order to keep things straight. A multi-INTG sequence can be in an answer state or it can be in an abort state. These states are controlled by using negative numbers in the GTMOD field of the control item. Once an instance is received which is not an ANSR, the multi-INTG sequence goes into an abort state and remains in that state until the last instance is finally received. If all instances arrive and all are ANSRs (there is no reason to suspect a mix of ANSRs and ABRTs in normal operation), then the logical transaction is considered satisfied. In either case, all instances, ABRTs and/or ANSRs, must be received before final processing can commence. If an instance arrives and is an ANSR and the sequence itself is still in an answer mode (i. e., no abnormal instances have been received yet), then the following processing is done. NETRESD first updates the control item (decrements the instance count) and then goes into the NEPOL sequence described above for single-INTG sequences. However, in this case, the hash function, NEHASH, is used to identify and remove duplicate records that may come across from the host. Duplicate records are more than likely to occur since each instance represents a separate disjunct of the dependent FIND statement's selection criteria. Hence there is no guarantee that the ORed sequence will not occasionally select the same record. After each NEPOL return (i. e., after it has isolated the next record-id), NETRESD calls NEHASH to determine if the record is unique. If it is not, NETRESD calls NEPOL to drop the current logical line and search for the next record-id. This effectively causes NEPOL to completely bypass the current record response. For each new record-id status from NEHASH, NETRESD updates the hit count in NERC and calls NEPOL with a write current logical line and search for next record-id status. In this case NEPOL writes out each logical line until the next record-id is encountered.

Following completion of the NEPOL sequence NETRESD then checks to see if this was the last instance of the multi-INTG sequence. If it is not, NETRESD then writes out a potential dirty record-id block (i.e., it may contain newly entered record-ids), the current state of the hash data, and the structure NERC. These are written out to eLTN. If the multi-INTG sequence is completed (i.e., all instances have come back), then NETRESD goes through the identical sequence described above for a single-INTG sequence, where TSOT and possibly DISPLAY processes are called.

NEPOL (Physical Response to Logical Response Processor for NETRESD) — This function is responsible for converting host responses (i.e., ANSR, ABRT FAULT or CAUSE) into a set of uniform length response lines. This version of NEPOL (see NEPOL for NETRESR) also provides a scan mechanism to be used jointly with the hashing logic for determining unique records via their record-ids. NEPOL first checks to see if a previous logical line must be written out and does so if required. Then NEPOL reads in the next (or first) response line from the host. Each character is scanned for a newline or new page character and, if a match is made, the logical line containing the saved characters is blank filled to ensure a constant logical line length. Then NEPOL determines whether it should search the logical line against a substring. If it does, it performs the search and returns with a zero status if the search was successful. If the search is unsuccessful, NEPOL checks to see if the line should be written out to cLTN and does so if required. NEPOL continues processing as specified above until the search is successful or the end of the response file is encountered, both of which cause NEPOL to return. If no search is to be performed, NEPOL processes lines as specified above until the EOF is encountered, at which time NEPOL returns.

NEHASH (Record-id Hash Processor) – This function provides the important facility to hash DIAOLS record-ids efficiently in order to eliminate duplicate records that may be selected during a multi-INTG sequence. Before discussing the data flow of NEHASH, some information must be supplied on the local data structure NERC. NERC is diagrammed below.



Generally, NERC contains the required information which reflects the current state of a multi-INTG sequence with respect to collected record responses from the DIAOLS host. The NERC structure itself contains the hash table, field NEHTAB; the next available record-id item specified in record-id blocks, field NEIDNAS; actual record-id size, NEACTSZ; record-id count (hence a running hit count for the interrogation), NERECNT; normalized record-id size to the nearest power of two, NERECSZ; and the element-short-name (as recognized on the DIAOLS host) of the record-id, NERNAM. NERC is stored in data file eLTN, and is followed by the actual list of collected record-ids. Processed records corresponding to these record-ids (via NEPOL) exist on the file cLTN.

NEHASH is recursive in nature and utilizes the UNIX random number generator (SRAND and RAND) in its operation. The initial call to NEHASH passes a negative one to indicate that this is the first call. NEHASH then initiates the random number generator and proceeds to hash the record-id value using the following algorithm. First, all characters of the record-id are exclusive-ORed together and the result is squared. NEHASH isolates the middle nine bits of the result to be used as an index into the hash table (i.e., the hash table is 512 words in length). This final result is then used to point into the hash table, NERC.NEHTAB, to see if the item is occupied. If the item in NERC.NEHTAB is empty (zero) then the record-id is new and NEHASH inserts the record-id into the next available record-id block, places the absolute pointer to the record-id into the hash table item, (i.e., the one pointed to by the hashed record-id), and then returns with a new record-id indicator (a positive one). If NEHASH finds that the hash table item is not empty, hence points absolutely to some other record-id, then it must retrieve this record-id and determine if they are indeed the same. If they are, then NEHASH returns with an old record-id indicator (a zero). Otherwise, NEHASH must resolve this collision by calling itself for a new hash value. For this case, NEHASH calls itself by passing the current hash value. The next invocation of NEHASH determines that this is not the first call for this particular record-id and then calls the random number generator, adding its result to the input hash value. The whole process is then repeated until either a new record-id or old record-id state is determined. NETRESD, the calling function for NEHASH, allows approximately 80 percent saturation of the hash table before truncating response data (this allows approximately 400 records).

<u>NECLNUP (Clean-up Processor)</u> – This function is responsible for removing most LTN related data files and finalizing GLIST and GTDES. This function is passed two parameters, the first pointing to the appropriate item in GLIST and the other a flag specifying whether the LTN was successful or unsuccessful (i.e., successful is equated to an ANSR condition on the query). First, NECLNUP removes all LTN files except for the

user output file, dLTN. Then the user's list file, GLIST, is locked and updated as follows. If the list is an active, explicit list (i.e., generated by a SAVE statement), it is set to a completed status. If the list is implicit, and/or the LTN is not successful (via the second parameter), then NECLNUP removes the list entry from GLIST and unlinks the associated GRMAP and GRDAT files. Next, GLIST is written out and unlocked. NECLNUP then sets the list ID field of the current GTDES item to zero, if the list is implicit, to signify that the LTN must have its authorization checked when the user peruses his output (this is done by the ADAPT I VIEW process which calls the TAS Access Authorization Process (AAP)). Finally, GTDES is locked, the current item is written out, GTDES is unlocked, and NECLNUP returns.

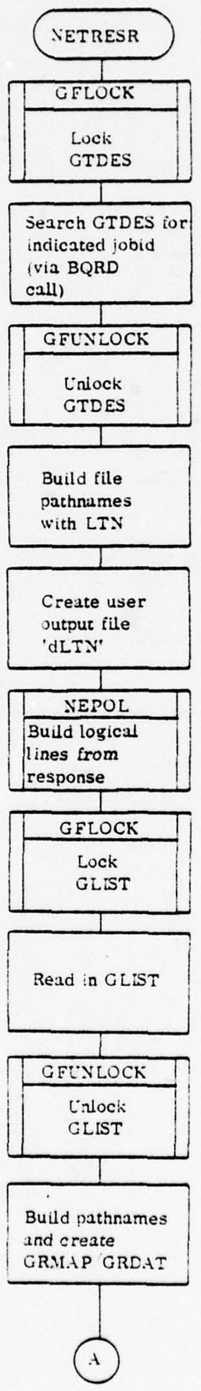

NETRESR
GFLOCK
Lock GTDES
Search GTDES for indicated jobid (via BQRD call)
GFUNLOCK
Unlock GTDES
Build file pathnames with LTN
Create user output file 'dLTN'
NEPOL
Build logical lines from response
GFLOCK
Lock GLIST
Read in GLIST
GFUNLOCK
Unlock GLIST
Build pathnames and create GRMAP GRDAT
A

Figure 42. NETRES Process Data Flow (Sheet 1 of 10)

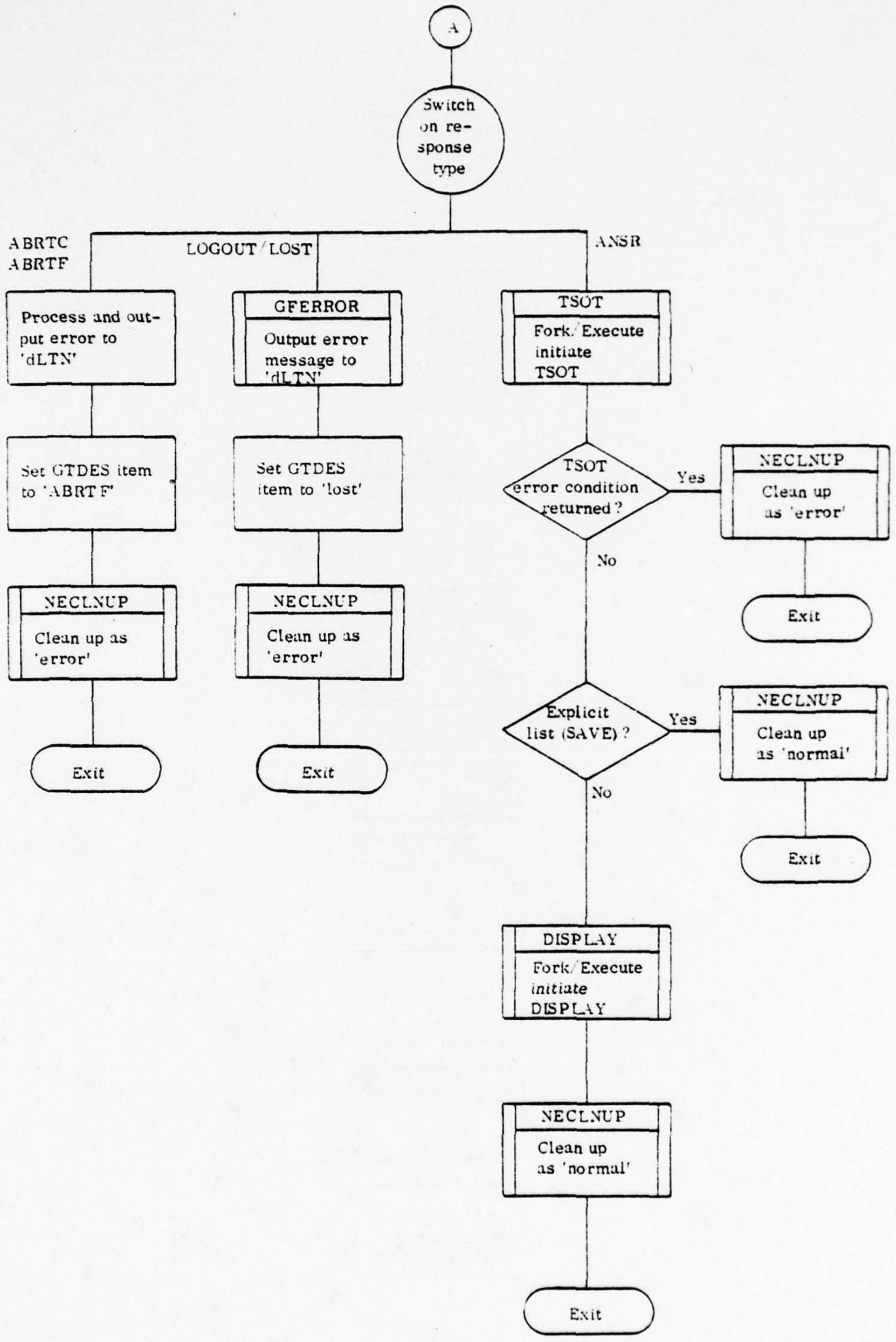


Figure 42. NETRES Process Data Flow (Sheet 2 of 10)

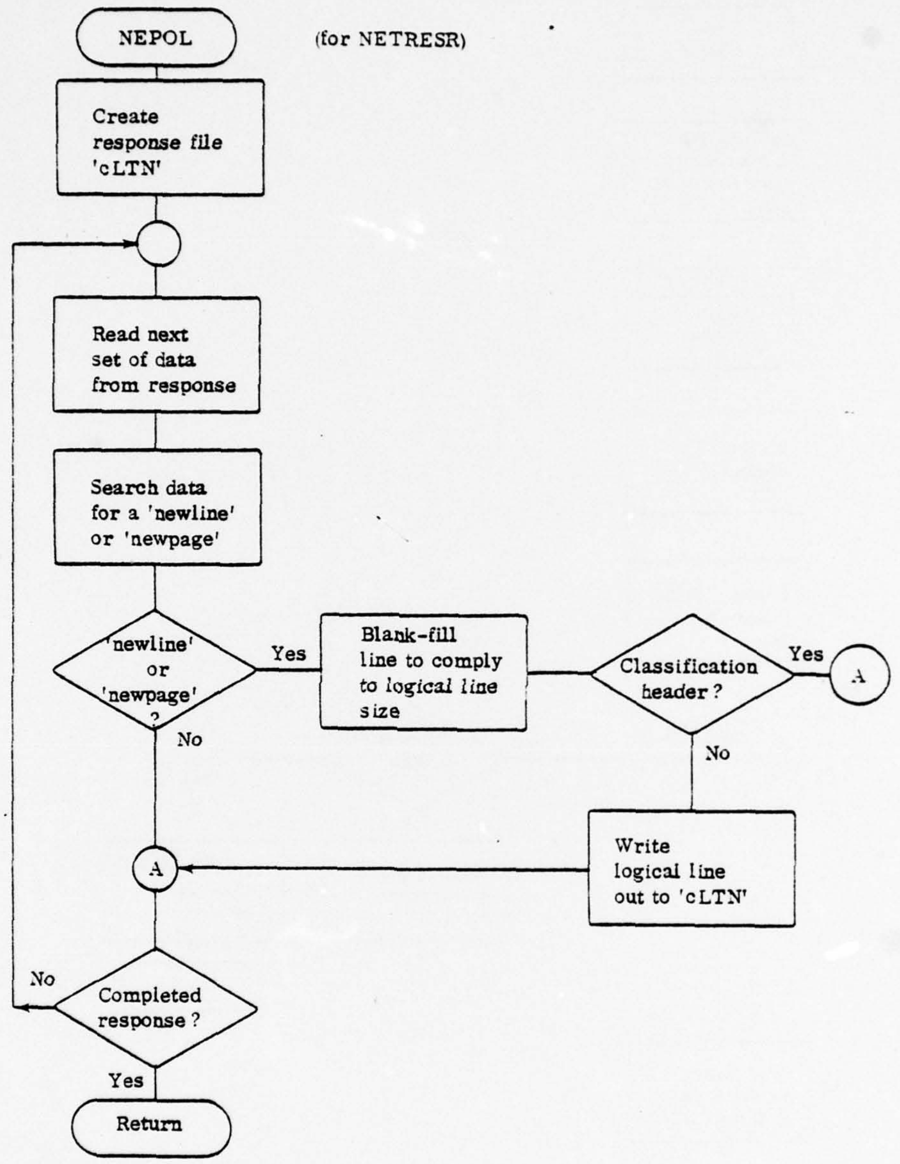

NEPOL
(for NETRESR)
Create response file 'cLTN'
Read next set of data from response
Search data for a 'newline' or 'newpage'
'newline' or 'newpage' ?
Yes
No
Blank-fill line to comply to logical line size
Classification header ?
Yes
A
No
Write logical line out to 'cLTN'
A
Completed response ?
No
Yes
Return

Figure 42. NETRES Process Data Flow (Sheet 3 of 10)

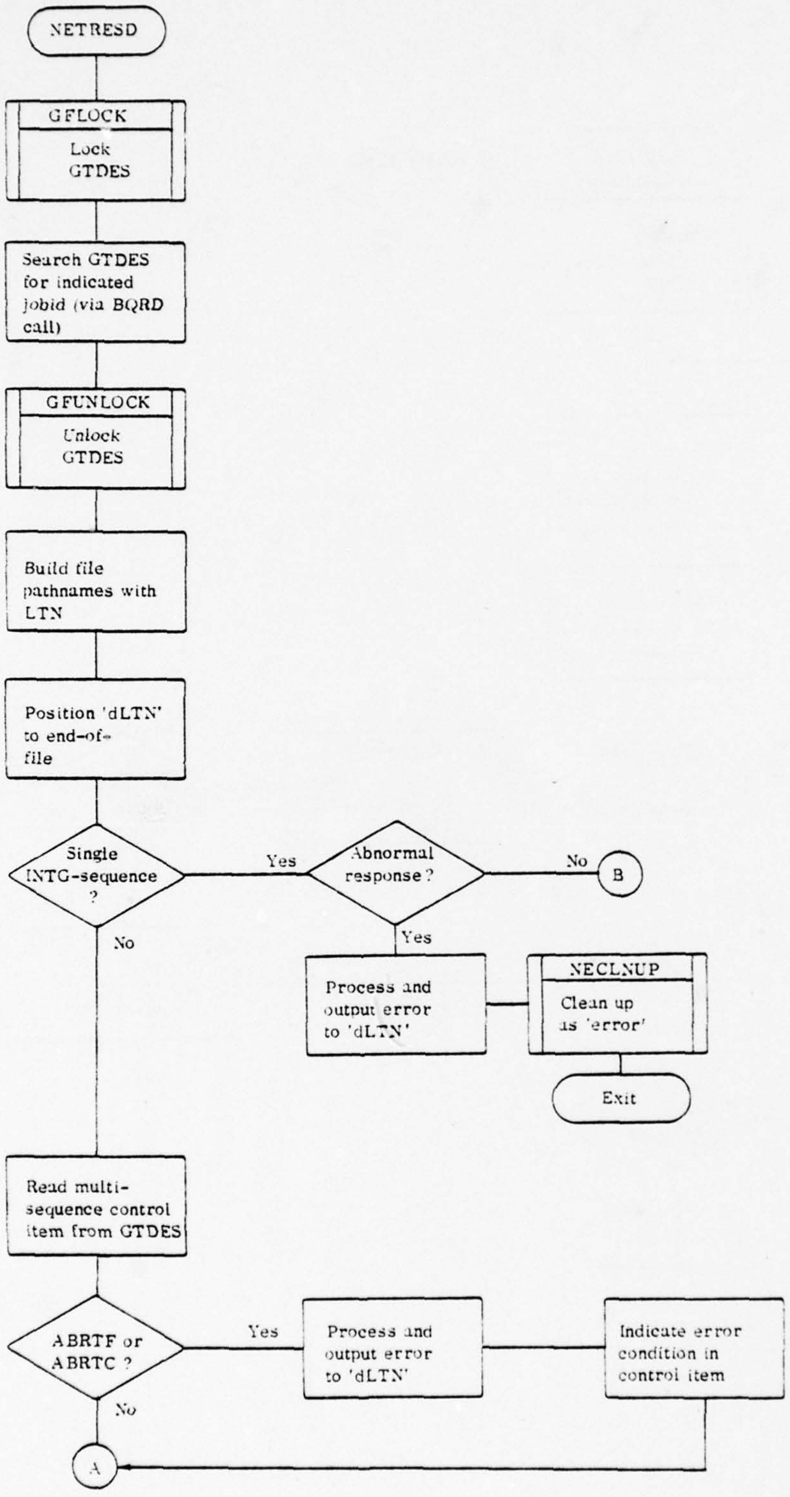

NETRESD
GFLOCK
Lock GTDES
Search GTDES for indicated jobid (via BQRD call)
GFUNLOCK
Unlock GTDES
Build file pathnames with LTN
Position 'dLTN' to end-of-file
Single INTG-sequence ?
Yes
No
Abnormal response ?
No
B
Yes
Process and output error to 'dLTN'
NECLNUP
Clean up as 'error'
Exit
Read multi-sequence control item from GTDES
ABRTF or ABRTC ?
Yes
Process and output error to 'dLTN'
Indicate error condition in control item
No
A

Figure 42. NETRES Process Data Flow (Sheet 4 of 10)

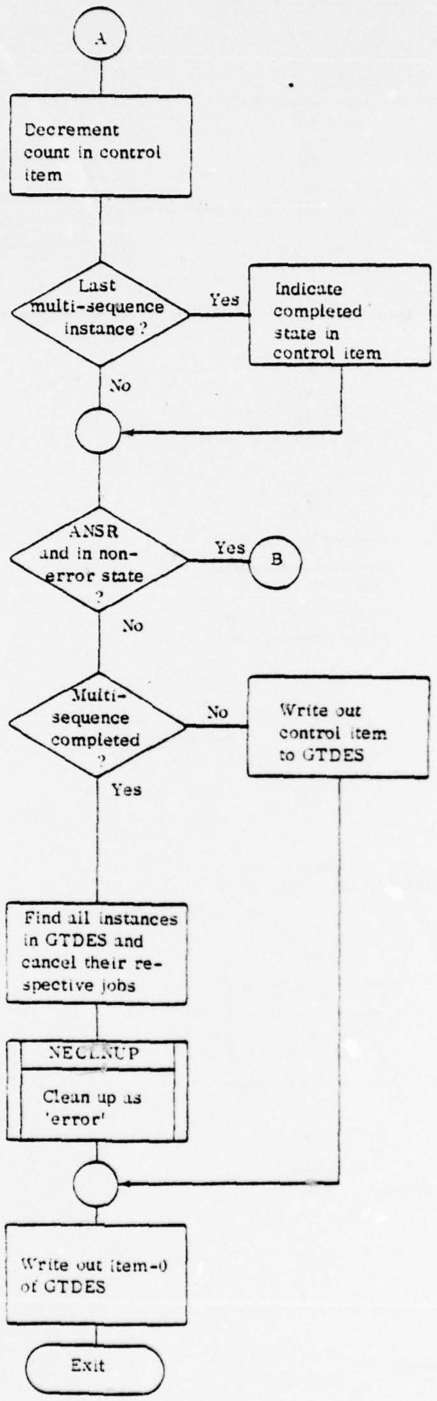

A
Decrement count in control item
Last multi-sequence instance ?
Yes
Indicate completed state in control item
No
ANSR and in non-error state ?
Yes
B
No
Multi-sequence completed ?
No
Write out control item to GTDES
Yes
Find all instances in GTDES and cancel their respective jobs
NECLNUP
Clean up as 'error'
Write out item-0 of GTDES
Exit

Figure 42. NETRES Process Data Flow (Sheet 5 of 10)

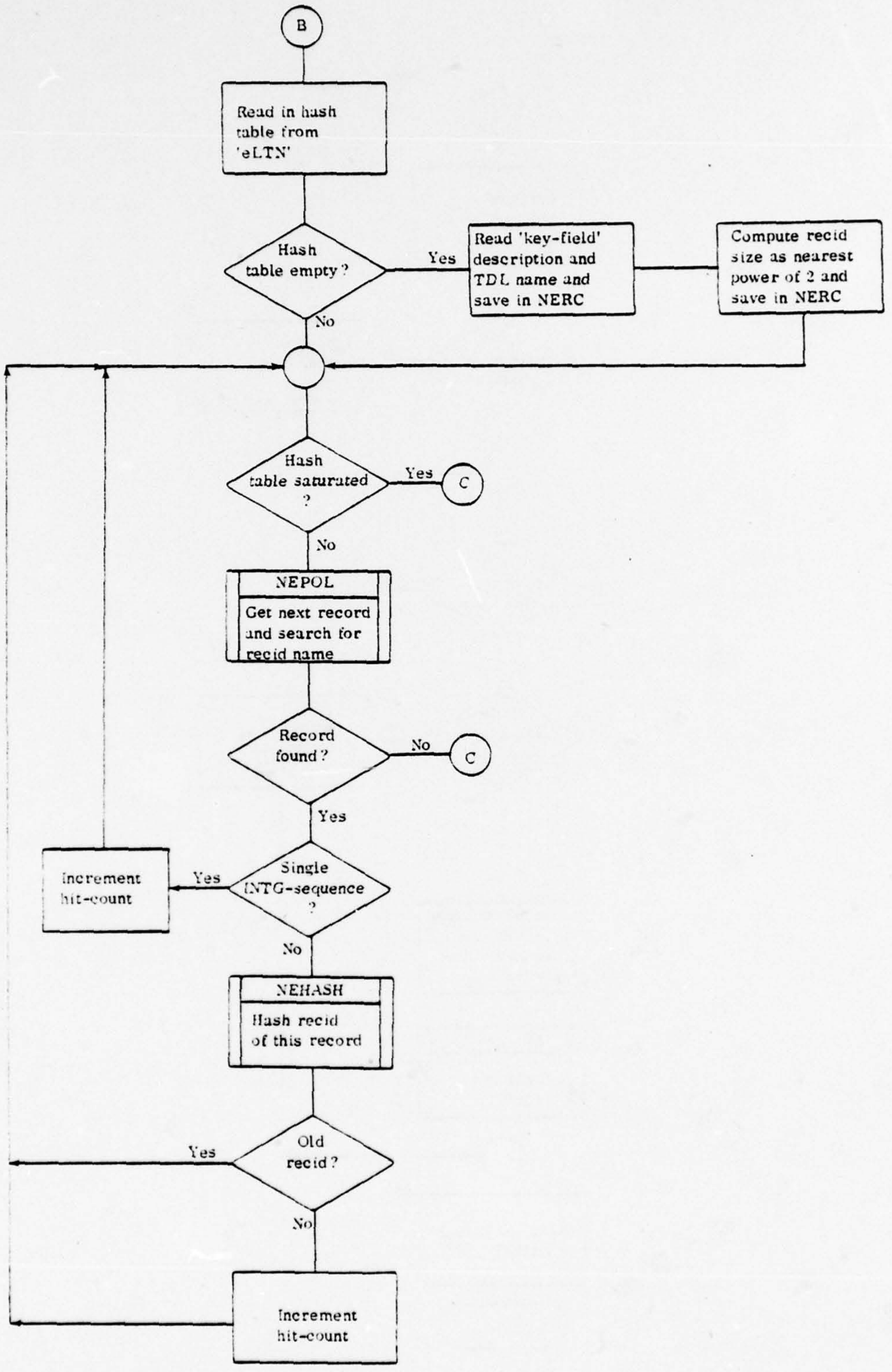

B
Read in hash table from 'eLTN'
Hash table empty?
Yes
Read 'key-field' description and TDL name and save in NERC
Compute recid size as nearest power of 2 and save in NERC
No
Hash table saturated?
Yes
C
No
NEPOL
Get next record and search for recid name
Record found?
No
C
Yes
Single INTG-sequence?
Yes
Increment hit-count
No
NEHASH
Hash recid of this record
Old recid?
Yes
No
Increment hit-count

Figure 42. NETRES Process Data Flow (Sheet 6 of 10)

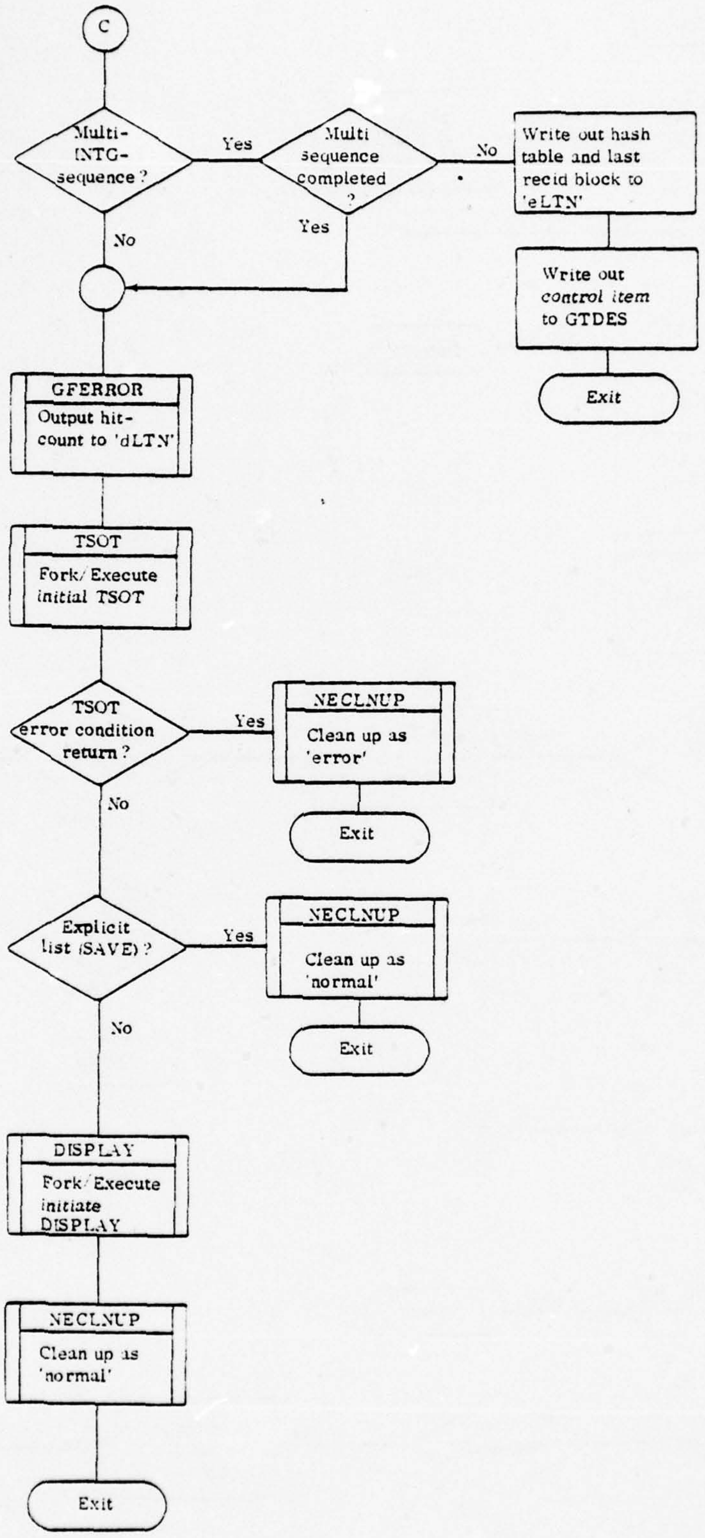

C
Multi-INTG-sequence ?
Yes
Multi sequence completed ?
No
Write out hash table and last recid block to 'eLTN'
Write out control item to GTDES
Exit
Yes
No
GFERROR
Output hit-count to 'dLTN'
TSOT
Fork/Execute initial TSOT
TSOT error condition return ?
Yes
NECLNUP
Clean up as 'error'
Exit
No
Explicit list (SAVE) ?
Yes
NECLNUP
Clean up as 'normal'
Exit
No
DISPLAY
Fork/Execute initiate DISPLAY
NECLNUP
Clean up as 'normal'
Exit

Figure 42. NETRES Process Data Flow (Sheet 7 of 10)

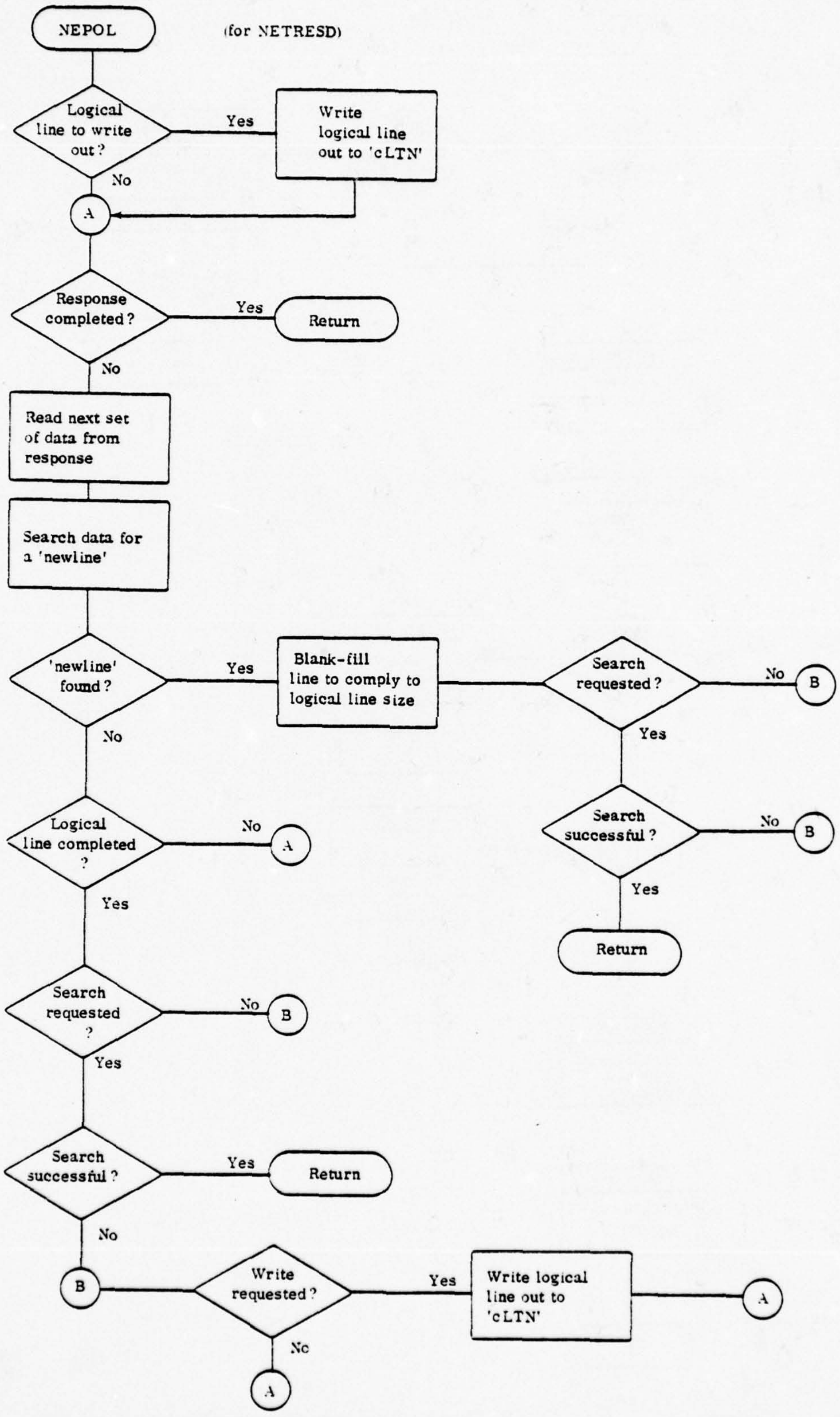


Figure 42. NETRES Process Data Flow (Sheet 8 of 10)

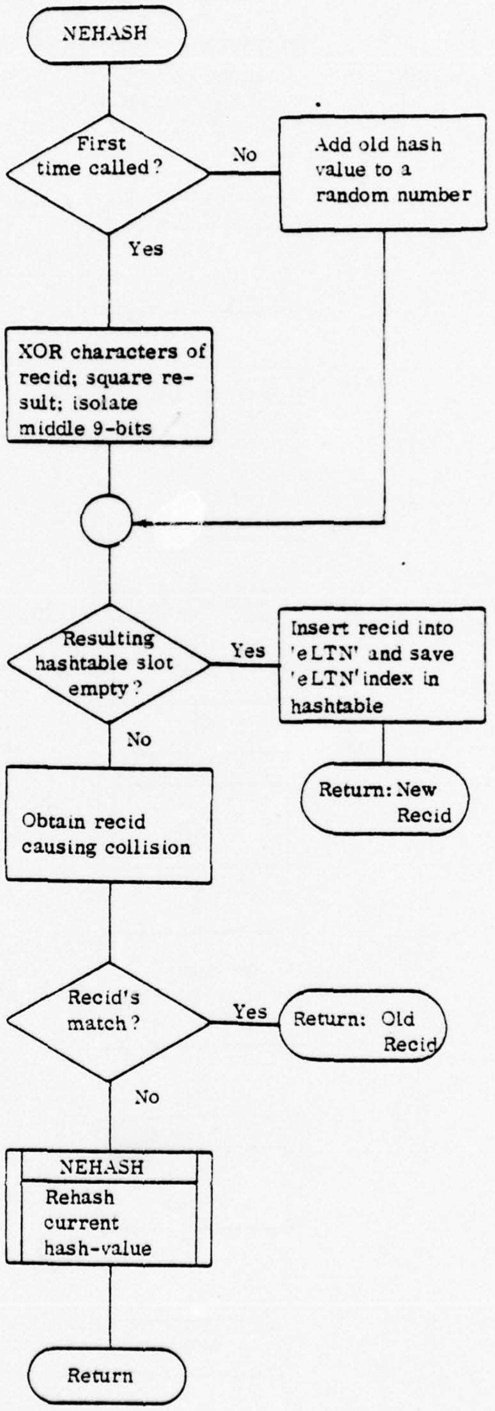

NEHASH
First time called?
No
Add old hash value to a random number
Yes
XOR characters of recid; square result; isolate middle 9-bits
Resulting hashtable slot empty?
Yes
Insert recid into 'eLTN' and save 'eLTN' index in hashtable
Return: New Recid
No
Obtain recid causing collision
Recid's match?
Yes
Return: Old Recid
No
NEHASH
Rehash current hash-value
Return

Figure 42. NETRES Process Data Flow (Sheet 9 of 10)

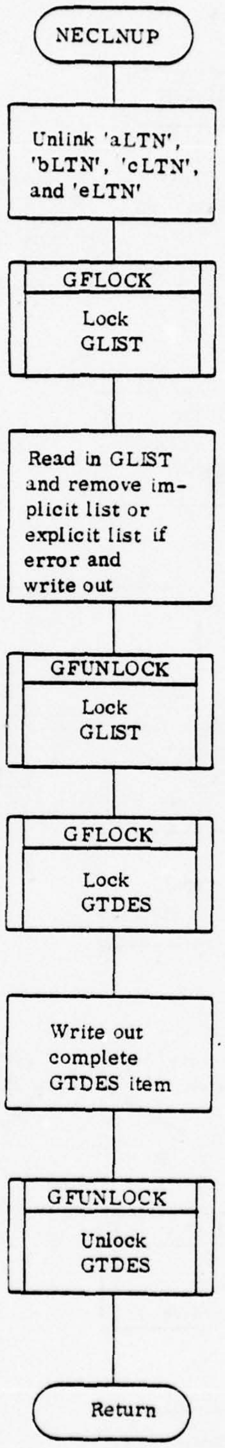

NECLNUP
Unlink 'aLTN', 'bLTN', 'cLTN', and 'eLTN'
GFLOCK
Lock GLIST
Read in GLIST and remove implicit list or explicit list if error and write out
GFUNLOCK
Lock GLIST
GFLOCK
Lock GTDES
Write out complete GTDES item
GFUNLOCK
Unlock GTDES
Return

Figure 42. NETRES Process Data Flow (Sheet 10 of 10)

## OPENCLOSE

The OPENCLOSE process validates and processes the OPEN and CLOSE commands. A file must be opened in order for queries to be made against that file.

GLOBAL DATA USAGE – OPENCLOSE uses the following global data structures:

| | | |
|---|---|---|
| GQFIL | – | batch query file names (TAS). |
| GTFLD | – | transformation file descriptions. |
| GTFLN | – | transformation file names. |
| GFILN | – | file names. |
| GTREE | – | parse tree. |
| GTDAT | – | parse tree data. |
| GOPEN | – | open files. |

LOCAL DATA USAGE – OPENCLOSE has no pertinent local data.

GENERAL PROCESS FLOW – The OPENCLOSE process is called whenever a user has input a UDL OPEN or CLOSE command. The EXEC passes to OPENCLOSE a user ID as well as the output file, GTREE, and GTDAT file descriptors. The following files are read: GOPEN, GTREE, GTDAT, GTFLD, GTFLN, and GFILN.

If the input statement is a CLOSE command without any filenames, the GOPEN file is set to zeroes and the file is written out, basically as a null file. If, however, the command has filenames specified, each name is validated to be an existing filename in GFILN. If it is not, an error is output and the next filename is processed. If the filename is valid and the statement is an OPEN command, the tranfile definition must exist and the file cannot already be opened. Next, the TAS Access Authorization Process (AAP) is called to determine whether both the user and terminal can access the target system, and, if batch, the specified file. If so, the file can be opened, its index is stored in GOPEN, and the next filename is processed. If the statement is a CLOSE command, the file index is located in GOPEN, the entry is removed, and the next filename is validated and processed. If at any time an error is detected, and error message is output and the next filename is processed.

Having processed all filenames, the OPENCLOSE process writes out the GOPEN file and returns to the EXEC. See figure 43 for data flow.

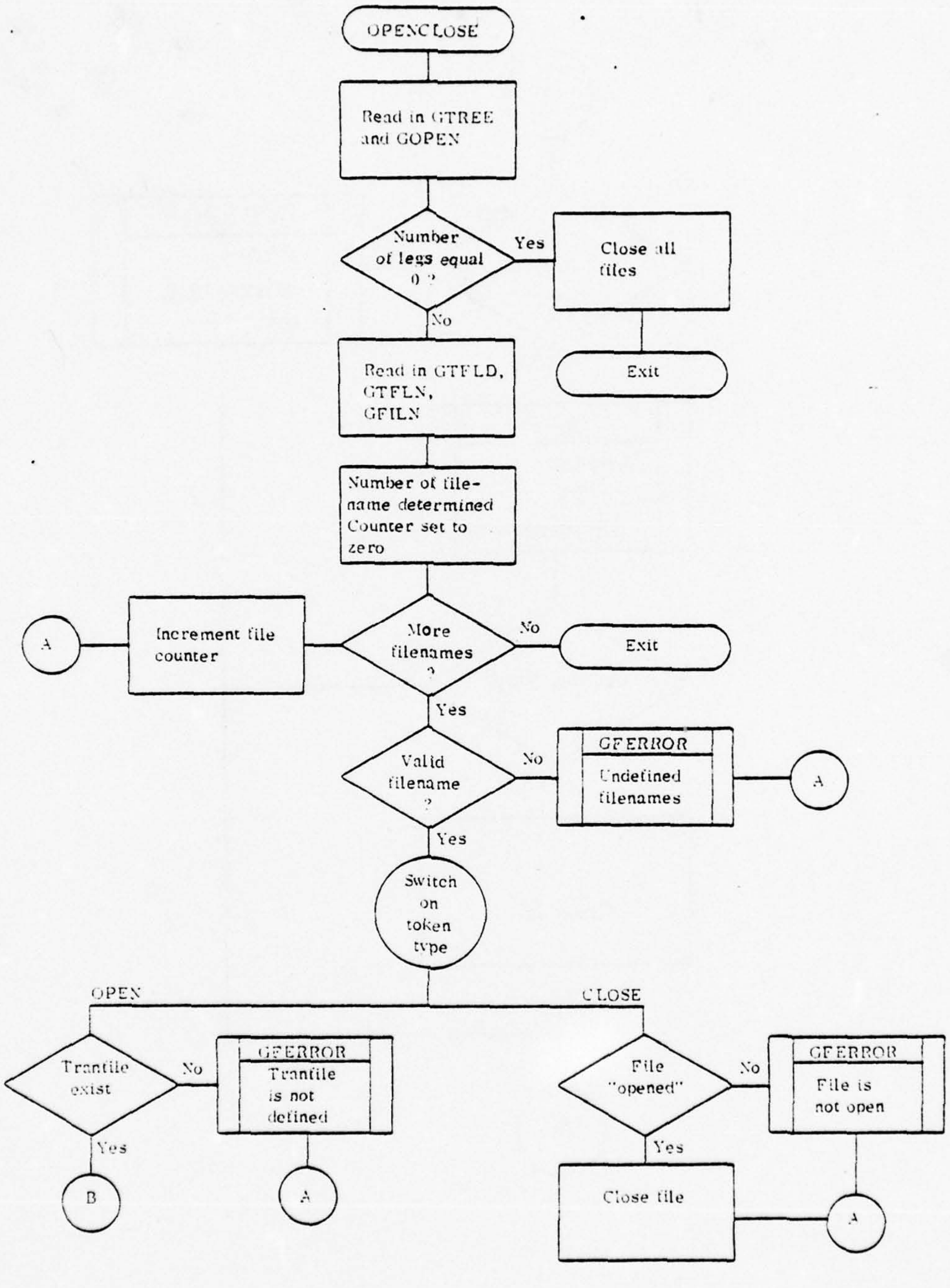


Figure 43. OPENCLOSE Process Data Flow (Sheet 1 of 2)

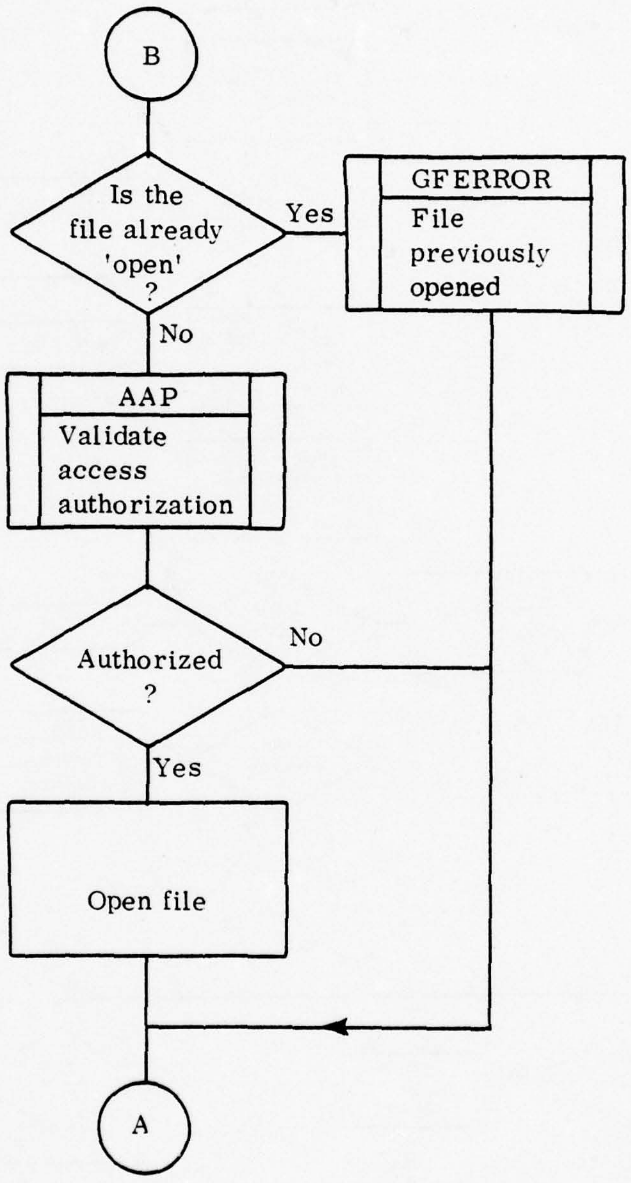


Figure 43. OPENCLOSE Process Data Flow (Sheet 2 of 2)

## QUIT

The QUIT process processes the UDL QUIT command. It modifies the user's status information and returns the user to the TAS command interpreter (Shell) program.

GLOBAL DATA USAGE – QUIT uses the following global data:

| | |
|---|---|
| GLABL | – label names and descriptions. |
| GOPEN | – open files. |
| GUSER | – user descriptions. |

LOCAL DATA USAGE – QUIT has no pertinent local data.

GENERAL PROCESS FLOW – The QUIT process is activated whenever a user has input a QUIT command. QUIT locks the ADAPT User Descriptions file (GUSER) via the global function GFLOCK, and then opens and reads the user's description. The user status is modified to indicate that this user is logged off, and GUSER is written and closed. QUIT then unlocks GUSER via the global function GFUNLOCK. Next, QUIT opens and reads this user's Label Descriptions file (GLABL). For each label which the user has utilized, that label's name is stored in the appropriate pathnames and the associated parse tree (GTREE) and parse tree data (GTDAT) files are unlinked. If the user has an open connection to an interactive target host system (also specified in GLABL), the TAS Interactive Query Interface (IQI) program which is responsible for that connection is sent a signal to close the connection. The user's GLABL file and Open Files file (GOPEN) are unlinked and QUIT exits. Since the QUIT process was activated by the EXEC via an execute (EXECL), its exit returns control to the next level of program, the TAS command interpreter (TASHELL). See figure 44 for data flow.

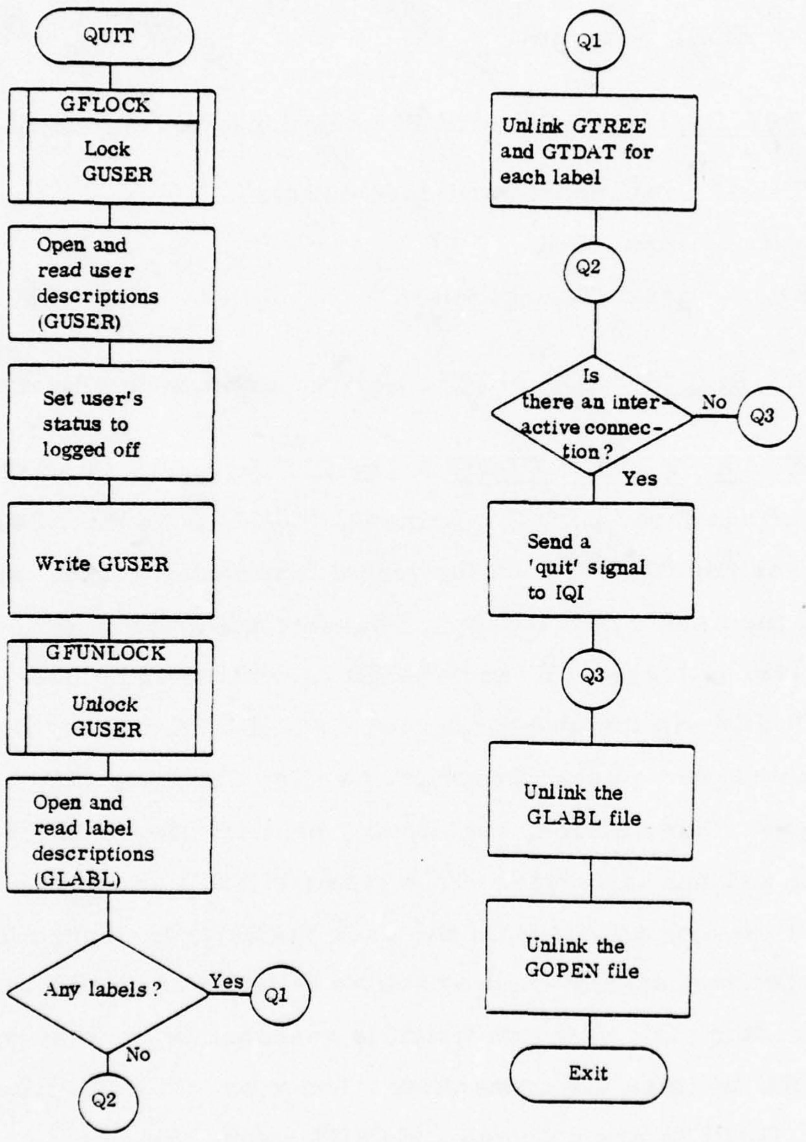

QUIT
GFLOCK
Lock GUSER
Open and read user descriptions (GUSER)
Set user's status to logged off
Write GUSER
GFUNLOCK
Unlock GUSER
Open and read label descriptions (GLABL)
Any labels ?
Yes
Q1
No
Q2
Q1
Unlink GTREE and GTDAT for each label
Q2
Is there an inter-active connec-tion ?
No
Q3
Yes
Send a 'quit' signal to IQI
Q3
Unlink the GLABL file
Unlink the GOPEN file
Exit

Figure 44. QUIT Process Data Flow

## TDL (Transformation Definition Language Process)

The TDL process organizes and stores the necessary UDL transformation specific information for files previously defined via DDL statements. While DDL provides the user with a facility for defining a UDL file's composition, TDL is the vehicle by which a user provides the specific information required to make transformations to target systems. The TDL process is responsible for the lexical and syntactic analysis, as well as actual statement processing of the TDL statement set.

<u>GLOBAL DATA USAGE</u> – TDL uses the following global data tables:

| | |
|---|---|
| GFILD | – file descriptions. |
| GFILN | – file names. |
| GTFLD | – transformation file descriptions (updated by TDL). |
| GTFLN | – transformation file names (updated by TDL). |
| GDBASE | – database names. |
| GANAM | – aggregate names. |
| GTADS | – transformation aggregate descriptions (created by TDL). |
| GFDES | – field descriptions. |
| GFNAM | – field names. |
| GTFDS | – transformation field descriptions (created by TDL). |
| GPFDS | – field position descriptions (created by TDL). |
| GTNAM | – transformation field and aggregate names (created by TDL). |
| GTREE | – parse tree. |
| GTDAT | – parse tree data. |

LOCAL DATA USAGE – The following local data are used by TDL:

| | |
|---|---|
| TDPOS | – position flag which is true if the file is position dependent; false, otherwise. |
| LXKYWRD | – keywords. |
| LXTOKEN | – keyword token type. |

GENERAL PROCESS FLOW – EXEC calls TDL whenever it encounters either a TRANFILE statement or an EXECUTE TRANFILE statement. The EXECUTE TRANFILE statement provides users with a mechanism to generate transformation files from TDL statements stored in an ordinary UNIX file. In contrast, the processing of a TRANFILE statement by EXEC implies an interactive environment with a user's terminal. If the validation mode flag is set via the MODE command when an EXECUTE TRANFILE or TRANFILE statement is recognized, the TDL statements are read by TDL for syntactic and semantic validation only.

The TRANFILE statement must appear once in the tranfile definition and must be the first statement of the definition. The following five pertinent bits of information may be contained in the tranfile statement.

a. The filename as known to UDL.

b. The filename as known to the host target system (necessary only if it differs from the UDL filename).

c. The target system on which the file resides.

d. The number of characters per record, if the file is position dependent.

e. The key field name, also file dependent information.

The filename is validated to be an existing UDL file with current data dictionary files and the transformation filename is contained in the character-constant which optionally follows. If, however, the character-constant is not given, the UDL filename is assumed to be the valid transformation filename. In either case, the transformation filename is stored in GTFLN.

The database identifier indicates the target system where the file resides, and this identifier is verified to be the name of a target system to which ADAPT I does transformations. If a file is position dependent, the integer following the keyword POSITION indicates the number of characters per record to be transmitted by the target system in response to a query. Finally, if appropriate, the key field name is read and verified to be a defined field name in the data definition file, GFNAM. The target system identifier, the position information, and the key field index are stored in GTFLD.

Having discussed the processing of the first TDL statement, two items should be mentioned. First, at the conclusion of processing any TDL statement, the LEX/parser is called upon to syntactically validate the next statement and to build the parse tree (GTREE) and the parse tree data (GTDAT). Secondly, whenever a syntactic or semantic user error is detected throughout the tranfile definition, an appropriate error message is output by the global function GFERROR to the user's terminal. After the TDL process encounters the first error, the following sequence of TDL statements is validated for context without any of the transformation files being created or written out. In other words, once an error is acknowledged, the TDL process operates only in the validation mode.

Field and aggregate transformation data are input by means of the tranfile aggregate statement and the tranfile field statement. The transformation aggregate statement consists of three parts: 1) a keyword which specifies the aggregate-type, RGROUP or ARRAY; 2) a name which is verified to be a locally established aggregate name in GANAM; and 3) the name by which the aggregate is known within the target system file. An aggregate may not be redefined within a transformation definition. If an aggregate is not included within the tranfile definition, the TDL process assumes the aggregate is known to UDL and the target system by identical names. Names are stored in GTNAM.

The tranfile field statement contains the following information:

a. The name of an existing field which must exist in GFNAM.

b. The conversion target-system name(s) for the field. If no discrepancy exists, the user may leave out the character-constant and TDL assumes the local field name to be correct. Names are stored in GTNAM.

c. The position integer is necessary if, and only if, the file is position dependent. This information is stored in GPFDS (field position description file).

d. The second integer, if included, indicates the number of occurrences of a multivalued field. The field-type, as stored in GFDES, must be multivalued or multivalued variable length. If the occurrence integer is suitable, the information is stored in GPFDS.

If a file is position dependent, all fields must be specified along with the position information. On the other hand, when a file is not position-dependent, a field need be referenced only if the local and target system names for the field differ.

The terminating statement of a transformation definition is the ETRAN statement. This statement initiates a series of checks for field and aggregate names which were left unmentioned in the tranfile definition. Again, all fields of a position dependent file must be referenced within the tranfile definition. The following files are created and written if no errors are encountered throughout the tranfile definition and the validation flag is not set:

| | |
|---|---|
| GTADS | – transformation aggregate descriptions. |
| GTFDS | – transformation field descriptions. |

GPFDS – field position descriptions.

GTNAM – transformation field and aggregate names.

Files GTFLD (transformation file descriptions) and GTFLN (transformation file names) are updated by the TDL process.

Control at this time is returned to EXEC to process the next input statement. See figure 45 for data flow.

## MAJOR FUNCTION DESCRIPTIONS

LXINIT (LEX/Parse Initialization) – LXINIT is merely an initialization function for the LEX/parse portion of this process. It initializes pertinent data, including a flag which it returns to the calling function to indicate EOF, syntax error, or statement accepted. LXINIT calls the parsing function (YYPARSE) and returns with the flag value.

YYPARSE (Parser) – YYPARSE is a parser which is produced by the YACC program running under UNIX. YACC produces the parser as well as a set of tables which the parser uses to organize the tokens passed to it by the lexical analyzer (YYLEX). These tables reflect the grammar of the TDL statements. YYPARSE calls YYLEX, which returns a value called a token type. If the token type is invalid according to the input statement syntax rules, an error message is output and YYPARSE calls YYLEX continuously until an end-of-input token type is returned. YYPARSE then returns with the return flag set to reflect a syntax error. If the token type is valid according to the input rules, the action specified in the YACC run is performed. The actions that can be invoked are: no action, build a parse tree node with the specified number of legs, store the final token type and write out the parse tree and parse tree data. After the action is performed, YYPARSE calls YYLEX for the next token type. When the end-of-input token type is received and its action performed, YYPARSE returns with the return flag set to reflect statement acceptance. If

YYLEX encounters an EOF, it returns an end-of-input token type and sets the return flag to reflect EOF.

YYLEX (Lexical Analyzer) – YYLEX, the function which actually inputs and processes each TDL statement, reads and saves each input character until a delimiter is encountered. The delimiters for ADAPT I TDL include: comma, newline (carriage return), space, left parenthesis, right parenthesis, and semi-colon. If an EOF is encountered, the return flag is set to reflect EOF and an end-of-input token type is returned. If a delimiter is encountered by itself, it is either passed to the parser (comma, left or right parenthesis, or semi-colon), or it is ignored. When a delimiter terminates a string of characters, YYLEX must determine what the characters represent. If the characters form a TDL keyword, YYLEX returns with that keyword's token type. Otherwise, YYLEX determines the type of constant which is formed by the characters (name, integer), stores the value of the constant in the parse tree data (GTDAT), stores a terminal node in the parse tree (GTREE), and returns with the constant's token type. If YYLEX encounters an error (e.g., too many characters input), it returns an invalid token type to the parser and thereby generates a syntax error message.

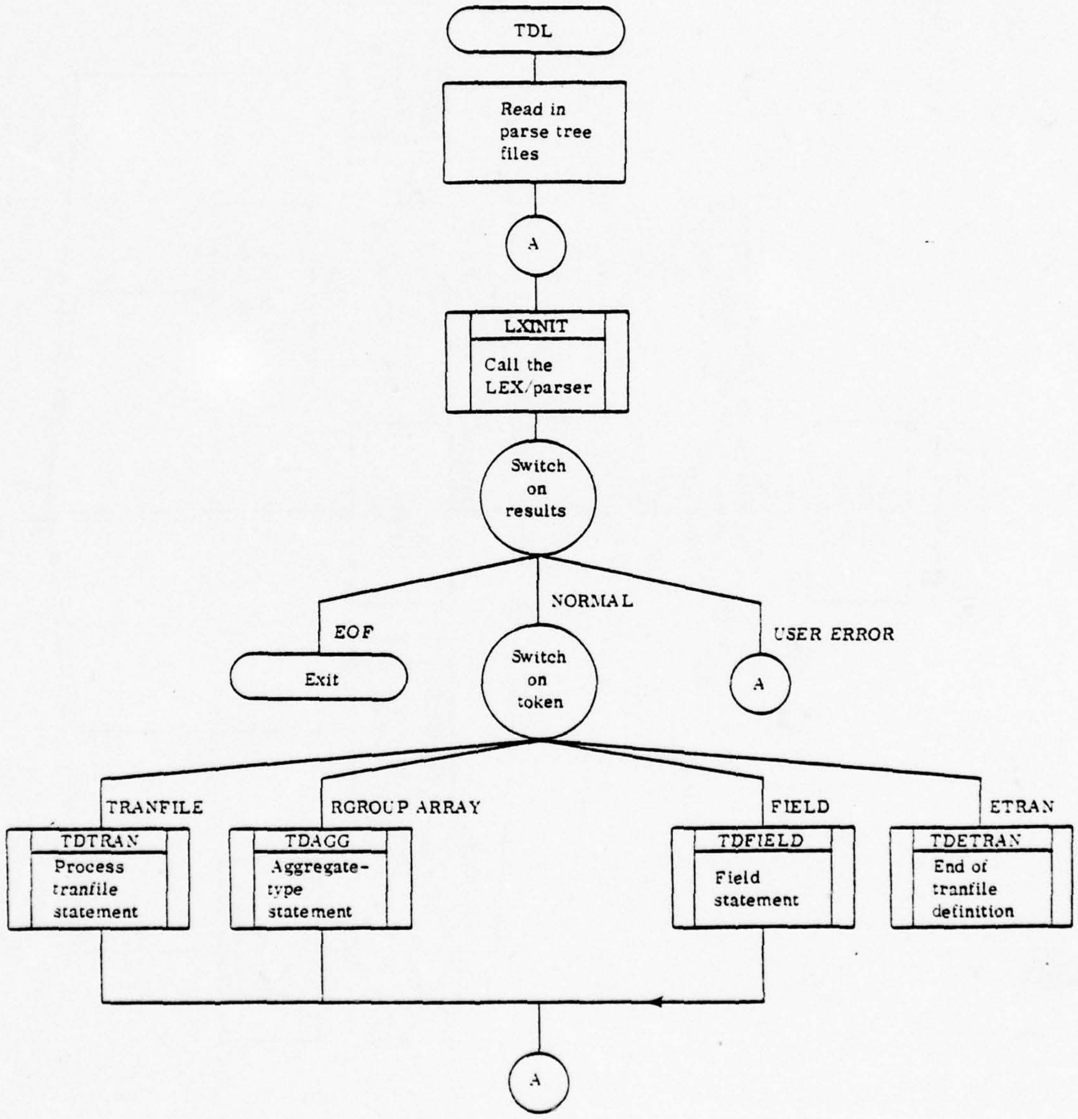


Figure 45. TDL Process Data Flow (Sheet 1 of 8)

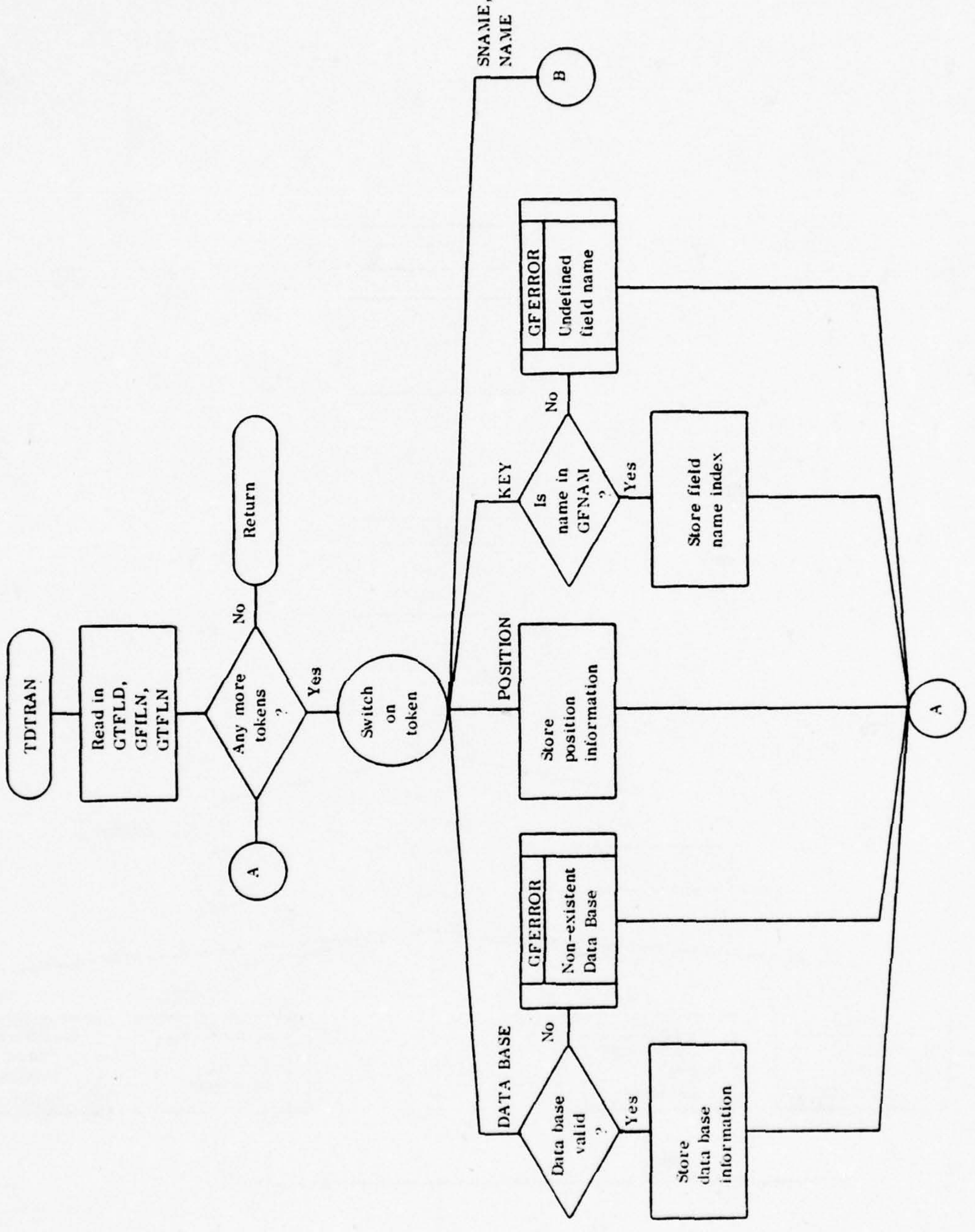


Figure 45. TDL Process Data Flow (Sheet 2 of 8)

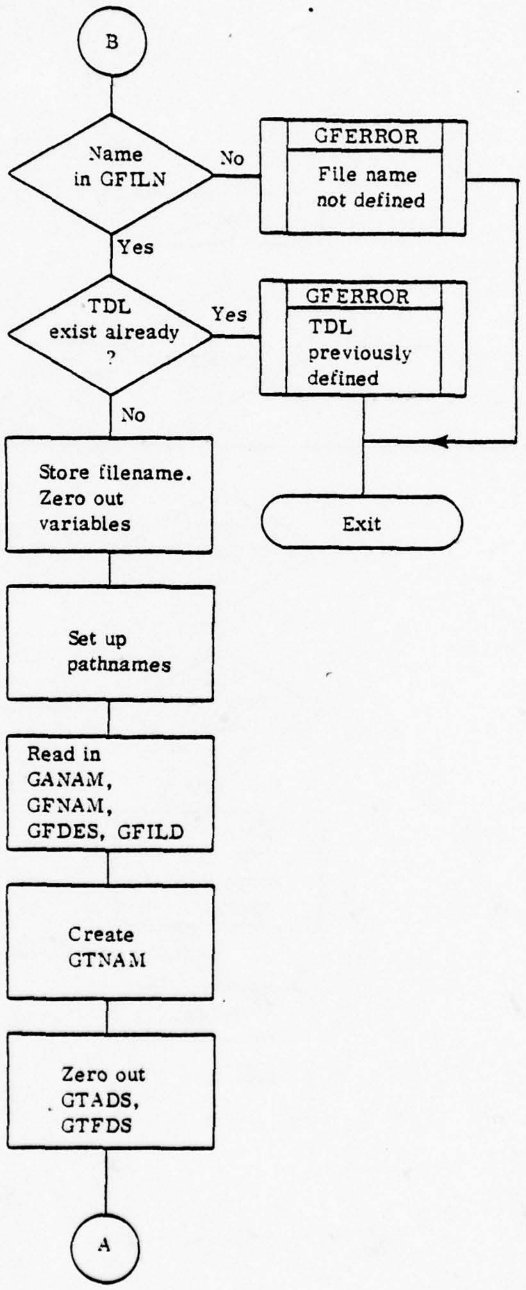


Figure 45. TDL Process Data Flow (Sheet 3 of 8)

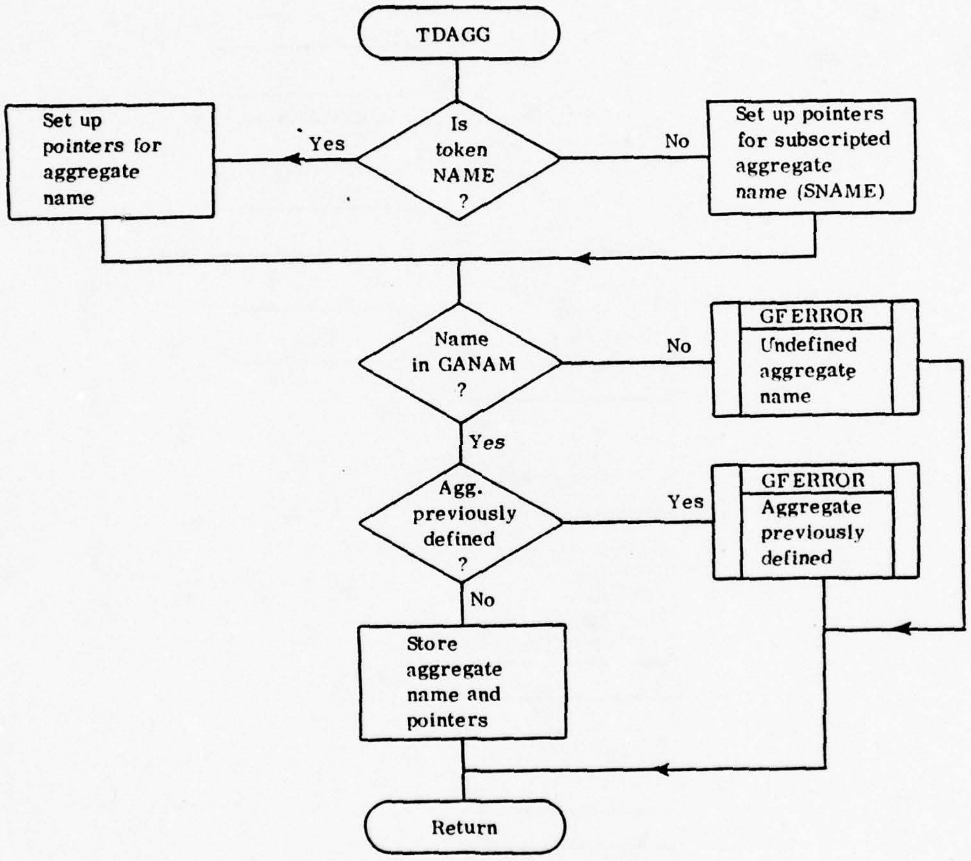

TDAGG
Is token NAME ?
Yes
No
Set up pointers for aggregate name
Set up pointers for subscripted aggregate name (SNAME)
Name in GANAM ?
No
GFERROR
Undefined aggregate name
Yes
Agg. previously defined ?
Yes
GFERROR
Aggregate previously defined
No
Store aggregate name and pointers
Return

Figure 45. TDL Process Data Flow (Sheet 4 of 8)

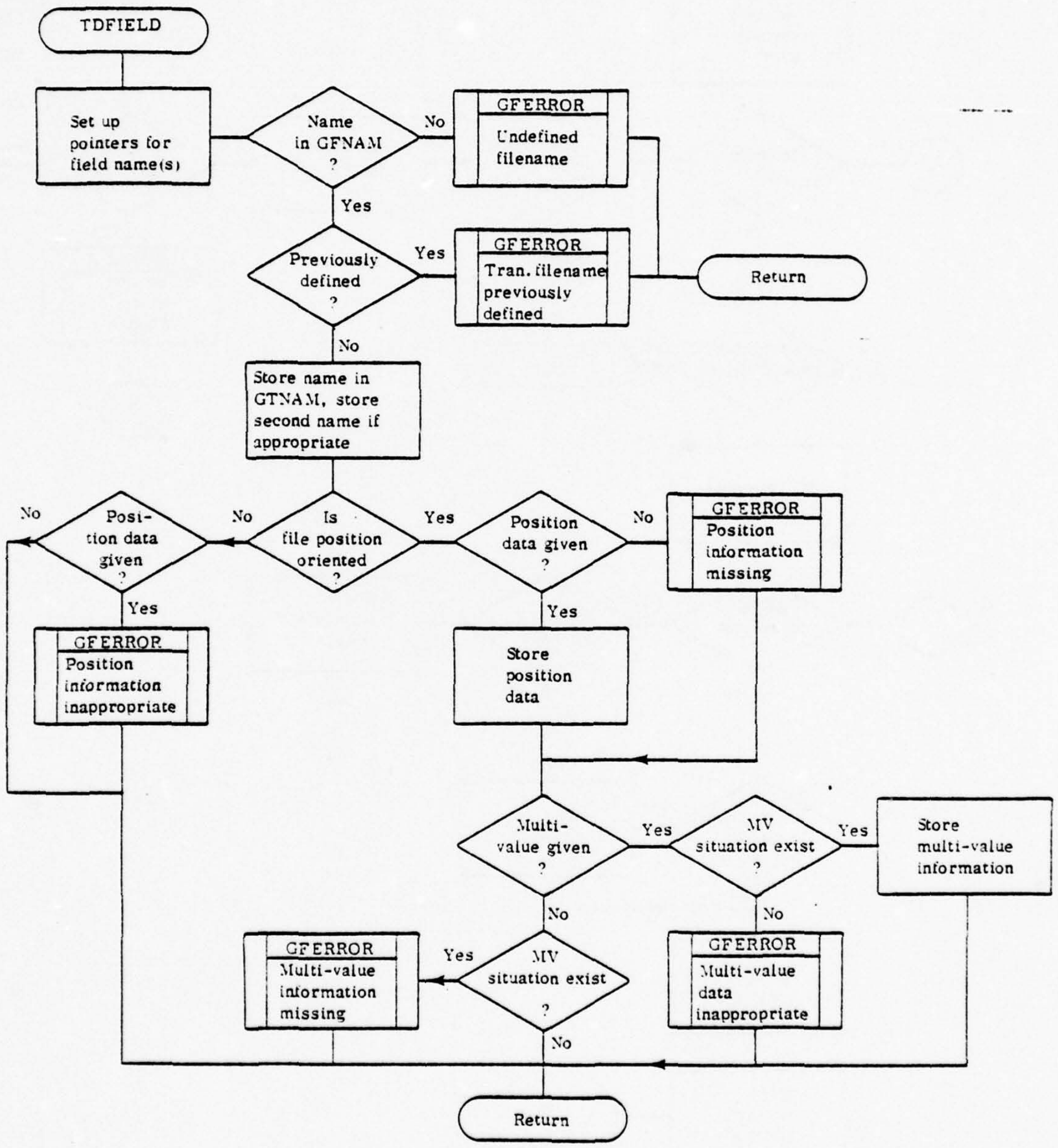

TDFIELD
Set up pointers for field name(s)
Name in GFNAM ?
No
GFERROR
Undefined filename
Yes
Previously defined ?
Yes
GFERROR
Tran. filename previously defined
Return
No
Store name in GTNAM, store second name if appropriate
No
Posi-tion data given ?
No
Is file position oriented ?
Yes
Position data given ?
No
GFERROR
Position information missing
Yes
GFERROR
Position information inappropriate
Yes
Store position data
Multi-value given ?
Yes
MV situation exist ?
Yes
Store multi-value information
No
No
GFERROR
Multi-value information missing
Yes
MV situation exist ?
GFERROR
Multi-value data inappropriate
No
Return

Figure 45. TDL Process Data Flow (Sheet 5 of 8)

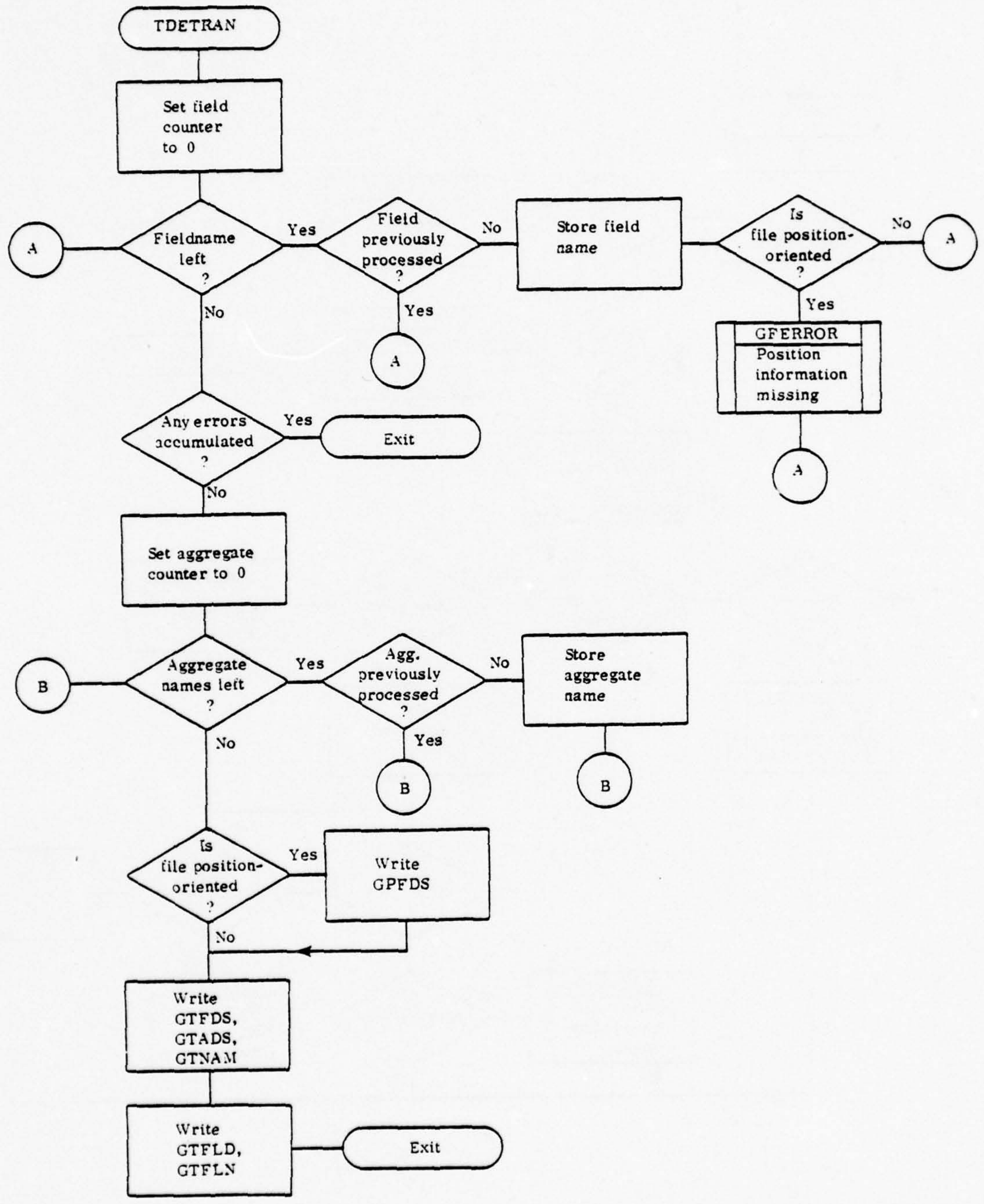


Figure 45. TDL Process Data Flow (Sheet 6 of 8)

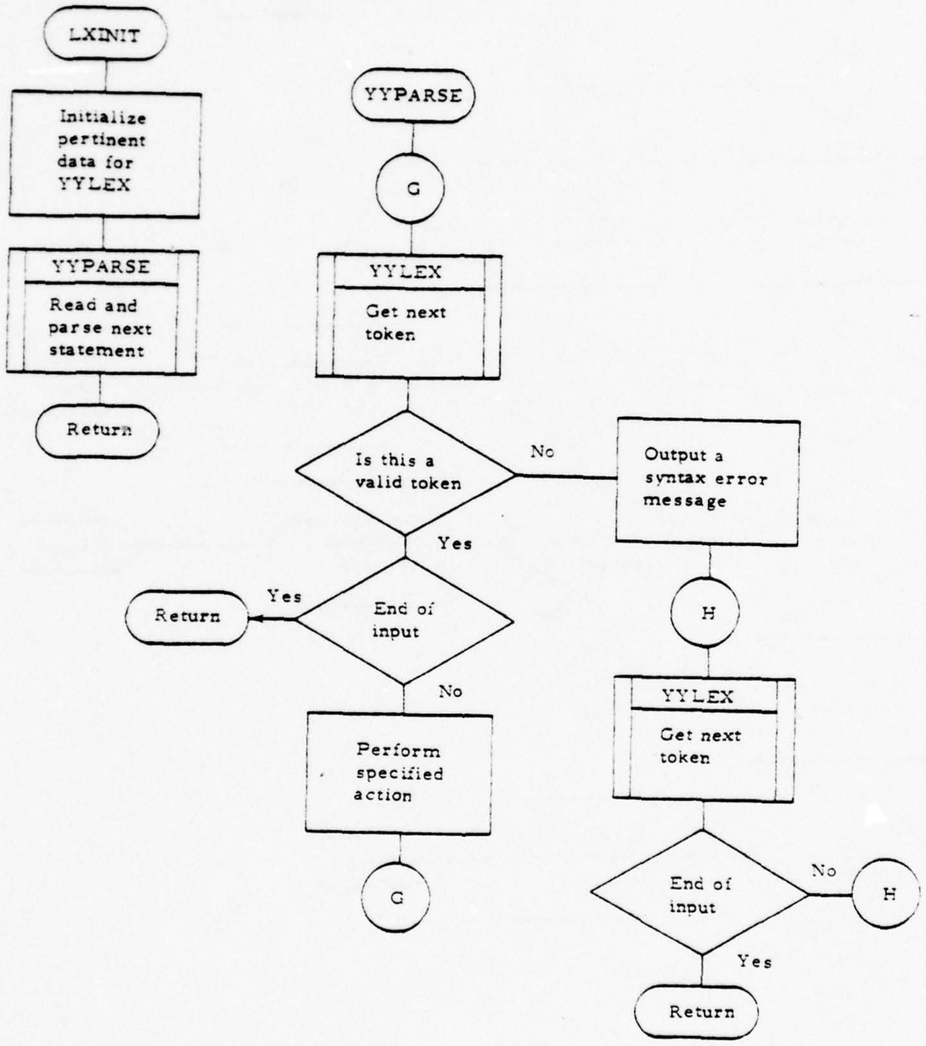


Figure 45. TDL Process Data Flow (Sheet 7 of 8)

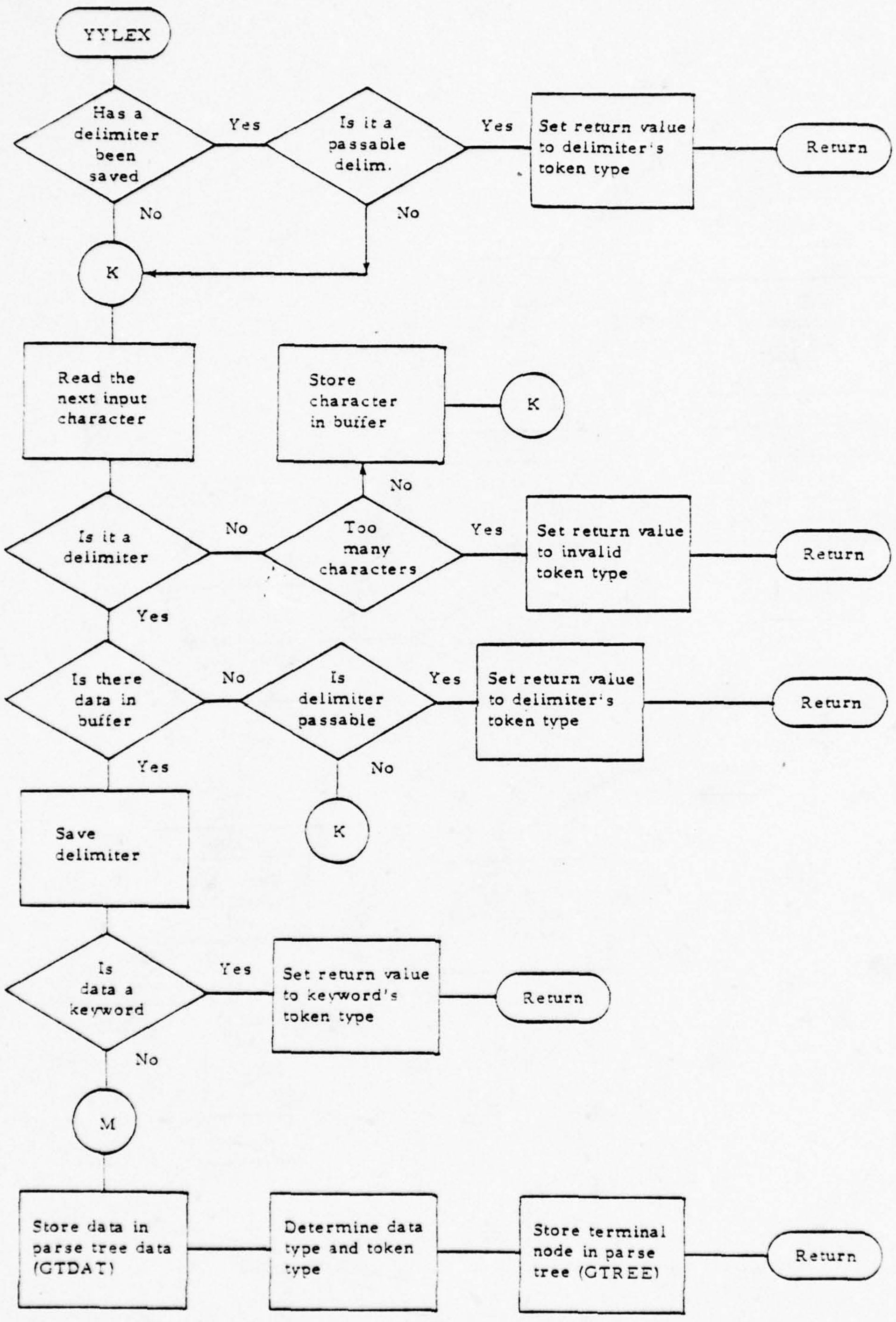

YYLEX
Has a delimiter been saved
Yes
Is it a passable delim.
Yes
Set return value to delimiter's token type
Return
No
No
K
Read the next input character
Store character in buffer
K
No
Is it a delimiter
No
Too many characters
Yes
Set return value to invalid token type
Return
Yes
Is there data in buffer
No
Is delimiter passable
Yes
Set return value to delimiter's token type
Return
Yes
No
K
Save delimiter
Is data a keyword
Yes
Set return value to keyword's token type
Return
No
M
Store data in parse tree data (GTDAT)
Determine data type and token type
Store terminal node in parse tree (GTREE)
Return

Figure 45. TDL Process Data Flow (Sheet 8 of 8)

## TSIGB (Target System Input Generator: Batch)

The TSIGB process is the ADAPT I process which controls all conversions of UDL statements into appropriate batch target system transformation strings. This process, plus NETRES, performs analogous functions to those provided by TSIGI.

GLOBAL DATA USAGE – The following ADAPT I global data structures are utilized by the TSIGB process.

| | |
|---|---|
| GTDES | – transaction descriptions. |
| GLABL | – label names and descriptions. |
| GPFDS | – field position descriptions. |
| GLIST | – list names and descriptions. |
| GTREE | – parse tree. |
| GTDAT | – parse tree data. |

LOCAL DATA USAGE – Refer to the discussion for this paragraph in TSIGI. Also refer to the description of the TSIGI decompiler, TIDECOMP.

GENERAL PROCESS FLOW – Under ADAPT I the TSIGB process is actually composed of two processes: TSIG1 for RYETIP and TSIG2 for DIAOLS. However, since both hosts are batch, the overall data flow of both processes is quite similar and is described here as a unit. Differences exist as host specific functions and are described below.

As is the case with TSIGI, the BEN process calls TSIGB for FIND statement processing and the VALIDATE process calls TSIGB for processing of DISPLAY or SAVE statements. FIND statement processing for batch systems is very simple. TSIGB just enters the FIND statement label into

GLABL and then returns to the EXEC process. As mentioned above, BEN calls TSIGB via an execute call, therefore, when TSIGB exits , it returns to the EXEC process which indirectly called it. FIND statement processing, with respect to transformations to the appropriate host query language, is not initiated until the user actually requests the output of some host data, via either the DISPLAY or SAVE statement. Therefore, transforms for the FIND statement are deferred until one of these statements is entered by the user. Upon entering a FIND statement for a batch host, the user only receives (assuming the statement is syntactically and semantically correct) a 'no hit count for a batch query' message on his terminal. When TSIGB receives a DISPLAY or SAVE statement, the processing is much more complicated. TSIGB calls the appropriate host specific function (TIRYEDS for RYETIP or TIISSDS for DIAOLS) to initiate statement processing. Since this is a batch interface both functions cause the generation of the appropriate host transforms representing the UDL statements. These transforms are registered under ADAPT I as a logical transaction via a Logical Transaction Number (LTN), and are sent to the appropriate host via the TAS process, BQRD. These transactions are also assigned a TAS unique JOBID in order to register them in TAS. The pathname of the appropriate response process (NETRESR for RYETIP and NETRESD for DIAOLS) is registered with BQRD for subsequent query response processing. Upon completion of this processing, TSIGB outputs the LTN to the user's terminal and returns to the EXEC process.

Although the upper data flow of both TSIGB processes is similar, the actual host specific logic is quite different. Due to the sophistication of the DIAOLS data structure/query language set, it is possible under UDL to compose a dependent FIND statement which can only be semantically satisfied via a set of INTG sequences, whereas with RYETIP, a single INTG sequence will always satisfy the transformation. See figure 46 for data flow.

## MAJOR FUNCTION DESCRIPTIONS

TIRYEDS (RYETIP DISPLAY and SAVE Processor) — This function is responsible for performing upper level processing of DISPLAY/SAVE statements for the RYETIP host. TIRYEDS first determines whether the statement is a DISPLAY statement. If it is, TIRYEDS generates an implicit list entry in the user's list file, GLIST. If not a DISPLAY statement, it is a SAVE statement and the appropriate explicit list entry is made in GLIST. Then TIRYEDS calls TIFORM to analyze the potential display-list. TIFORM will return a zero status if the user is requesting more than 70 percent of each selected record and TIRYEDS will copy the original position data files, GPFDS onto eLTN. If not, TIFORM has already copied the new GPFDS onto eLTN. Now TIRYEDS calls the decompiler, TIDECOMP, to generate the appropriate transforms which are written out to cLTN. Then TIRYEDS establishes a GTDES item (which it must lock since it is update sensitive), which it builds. Included in this item is the TAS JOBID (via a call to GFJOBID), the LTN, the file ID, and the list ID. TIRYEDS now calls the TAS Batch Query and Response Dispatcher (BQRD) process to register the INTG with TAS and to send it to the RYETIP host. TIRYEDS establishes the NETRESR process as the recipient process for any subsequent response from the host. Following the return from BQRD, TIRYEDS removes the transforms file, cLTN.

TIFORM (RYETIP Display-List Processor) — This function is responsible for determining if the user is requesting less than 70 percent of the entire record. If the user is requesting less than 70 percent of the record, a new position data file for this query is generated (GPFDS) by TIFORM. First, TIFORM checks to see if this is either a SAVE statement or a DISPLAY statement without a display-list. If either is the case, TIFORM returns with a zero status. This implies that the user is requesting more than 70 percent of the record (in this situation, the entire record). If the statement is a DISPLAY statement and a display-list is present, TIFORM must analyze the display-list, accumulating the number of characters being requested for output. For each field referenced in the display-list, TIFORM first checks to see whether it overflows a single RYETIP ANSR line by its size or, if multivalued, by its size multiplied by the number of allowable occurrences. If the field overflows a single ANSR line, TIFORM returns with a status of zero (i. e., it is not possible to generate transforms using FORMAT:PART where the data exceeds one line). If the field does not exceed one ANSR line, its size is added to the accumulated field sizes. For each iteration of the above sequence TIFORM checks to see if the accumulated size exceeds one line. If it does, the pass counter is incremented and the accumulated field size count is zeroed out. For each field processed, a new GPFDS position is generated along with the associated pass count. Upon the completion of processing the display-list, TIFORM checks to see if the total size exceeds 70 percent of the record size. If it does, it returns with a zero status. If not, the new GPFDS data file is written out to eLTN which will be used by the appropriate TSOT process when the response (ANSR) comes back from the host. Then TIFORM returns with a positive one status. Note, this new position file is used by a special decompiling function in generating the proper FORMAT:PART values.

TIISSDS (DIAOLS DISPLAY/SAVE Processor) — This function is responsible for providing upper level processing of DISPLAY/SAVE statements for the DIAOLS host. TIISSDS first determines whether an implicit list (DISPLAY) or an explicit list (SAVE) entry is required in GLIST. It then establishes a GTDES item for the INTG. At this point, TIISSDS does not know whether a multi-INTG sequence is required. If it is, then the GTDES item will become the multi-INTG sequence control item; otherwise, it will be used as the single-INTG GTDES item representing the INTG. If the statement is not a dependent FIND statement, TIISSDS calls the decompiler, TIDECOMP, to generate the appropriate transforms for this statement and then calls TIINTGF to generate the DISPLAY/SAVE transforms (via TIDECOMP) and issue the INTG via the TAS process BQRD. Finally, TIISSDS returns to TSIGB. If the FIND statement is dependent, TIISSDS calls TIDANAL to decompose the selection criteria, by disjunct, into separate INTG sequences. Then TIISSDS checks to see if normal disjuncts exist (disjuncts sans geographic or scoped rel-terms). If they do, a separate INTG representing all of them must be generated. First, TIISSDS must generate the original FIND statement transforms via TIDECOMP and then all the other normal disjuncts via a parenthetical AND operator. Finally, TIINTGF is called to generate the DISPLAY/SAVE transforms and to initiate the INTG. Then TIISSDS returns to TSIGB.

<u>TIDANAL (DIAOLS Decomposition Processor)</u> — This function is responsible for decomposing dependent selection criteria into separate manageable disjuncts for separate INTG generation. It is only called for dependent FIND statement processing. TIDANAL is composed of two separate phases. The first phase analyzes the selection criteria isolating separate disjuncts (as ordered by the BEN process). Each disjunct is a separate invocation of TIDANAL, hence TIDANAL is recursive. For each invocation of TIDANAL, the disjunct is grouped into three types of relation terms: geographic rel-terms (i. e., rel-terms whose relation operator is ALONG, INSIDE, or OUTSIDE); SCOPED rel-terms (i. e., a set of ANDed rel-terms inside a repeating group scoped operation; the expression is in scope-normal-form); and all other rel-terms, termed here as normal rel-terms. During this process it is also noted whether the rel-term is NOTed. Upon completion of this phase, TIDANAL then goes into the second phase. In the second phase, TIDANAL checks to see if the just processed disjunct contained only normal rel-terms. If it did, a pointer to the top of the disjunct (into GTREE) is saved in an external array for

later processing by TIISSDS. If the disjunct contains at least one geographic or scoped rel-term, TIDANAL proceeds to generate the appropriate transforms. First TIDECOMP is called to generate the transforms for the independent FIND statement. It also ascertains if normal rel-terms are present. If they are, the transforms must provide a parenthetical ANDed expression for their inclusion. Then for each geographic rel-term (if present) the appropriate transforms are generated by repeated calls to TIDECOMP. Then repeated calls to TIDECOMP are made for each SCOPE rel-term that may be present in the dependent FIND statement. Finally, TIDANAL calls TIINTGF to generate the final DISPLAY/SAVE transforms (via TIDECOMP) and to issue the INTG via a BQRD call. Note, for each disjunct contained in the dependent FIND statement, a separate INTG must be sent to the DIAOLS host. Since TIDANAL is recursive, it will continue returning to itself after each disjunct is processed until the final disjunct, after which it will return to the calling function, TIISSDS.

<u>TIINTGF (DIAOLS INTG Processor)</u> – This function is responsible for sending INTGs to the DIAOLS host via calls to the TAS process BQRD. It also controls multi-INTG bookkeeping for GTDES entries. TIINTGF first reestablishes the parse trees, GTREE and GTDAT, for the DISPLAY/SAVE statement. Following this, the decompiler TIDECOMP is called to generate the appropriate transforms for the current UDL statement. Then TIINTGF checks to see if this is a multi-INTG sequence. If it is, a new GTDES item must be established for this particular INTG instance. This GTDES item is given a subtransaction number, its own TAS JOBID, and is associated with the same LTN as assigned to the other INTG instances. That is, it is still associated with a single ADAPT I query. If the query is not a multi-INTG sequence, TIINTGF must only call the BQRD process to establish the new INTG. This is also done for the multi-INTG sequence after the above described processing has been accomplished. Following the BQRD call, TIINTGF reestablishes the transform file, cLTN, as a

zero length file and reinitializes the global decompiling variables (i.e., the decompiler may be called many times to satisfy a potential multi-INTG sequence query) and then returns to the calling function.

TIDECOMP (Decompiler) – Refer to the description of TIDECOMP in the TSIGI process.

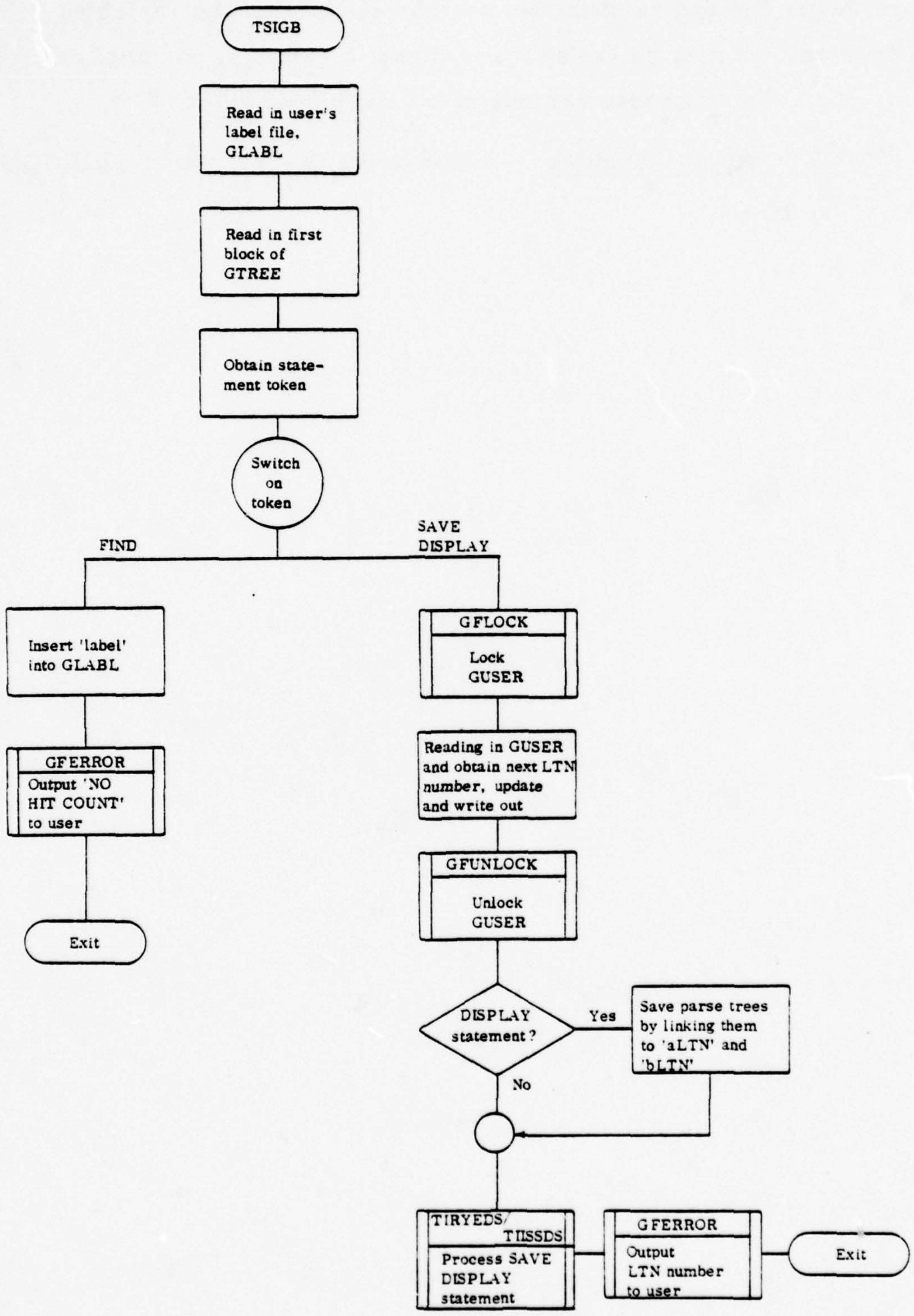

TSIGB
Read in user's label file, GLABL
Read in first block of GTREE
Obtain statement token
Switch on token
FIND
SAVE DISPLAY
Insert 'label' into GLABL
GFERROR
Output 'NO HIT COUNT' to user
Exit
GFLOCK
Lock GUSER
Reading in GUSER and obtain next LTN number, update and write out
GFUNLOCK
Unlock GUSER
DISPLAY statement?
Yes
No
Save parse trees by linking them to 'aLTN' and 'bLTN'
TIRYEDS/ TIISSDS
Process SAVE DISPLAY statement
GFERROR
Output LTN number to user
Exit

Figure 46. TSIGB Process Data Flow (Sheet 1 of 7)

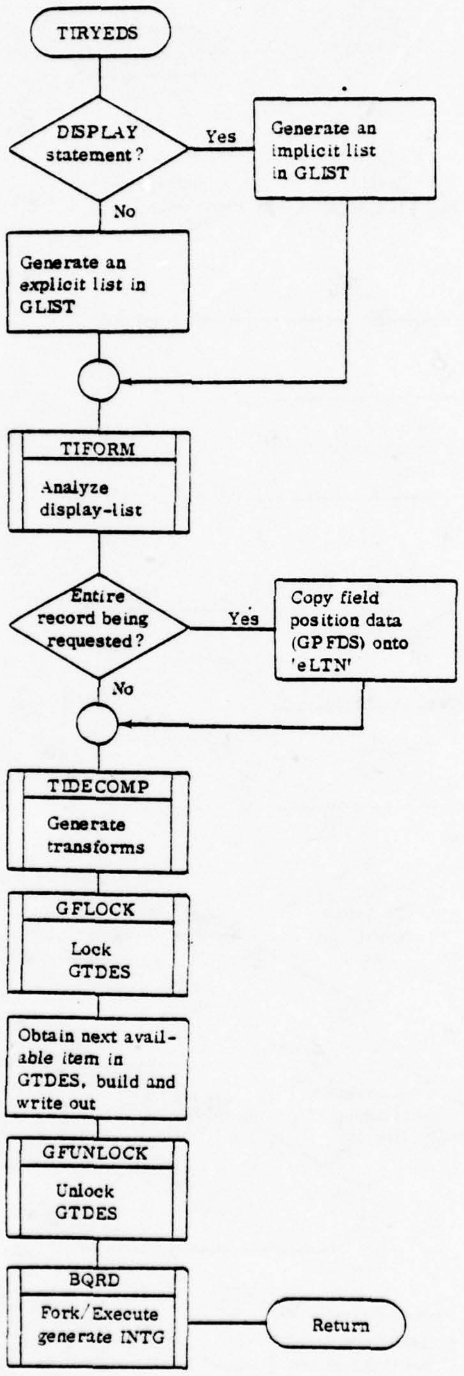

TIRYEDS
DISPLAY statement?
Yes
No
Generate an implicit list in GLIST
Generate an explicit list in GLIST
TIFORM
Analyze display-list
Entire record being requested?
Yes
No
Copy field position data (GPFDS) onto 'eLTN'
TIDECOMP
Generate transforms
GFLOCK
Lock GTDES
Obtain next available item in GTDES, build and write out
GFUNLOCK
Unlock GTDES
BQRD
Fork/Execute generate INTG
Return

Figure 46. TSIGB Process Data Flow (Sheet 2 of 7)

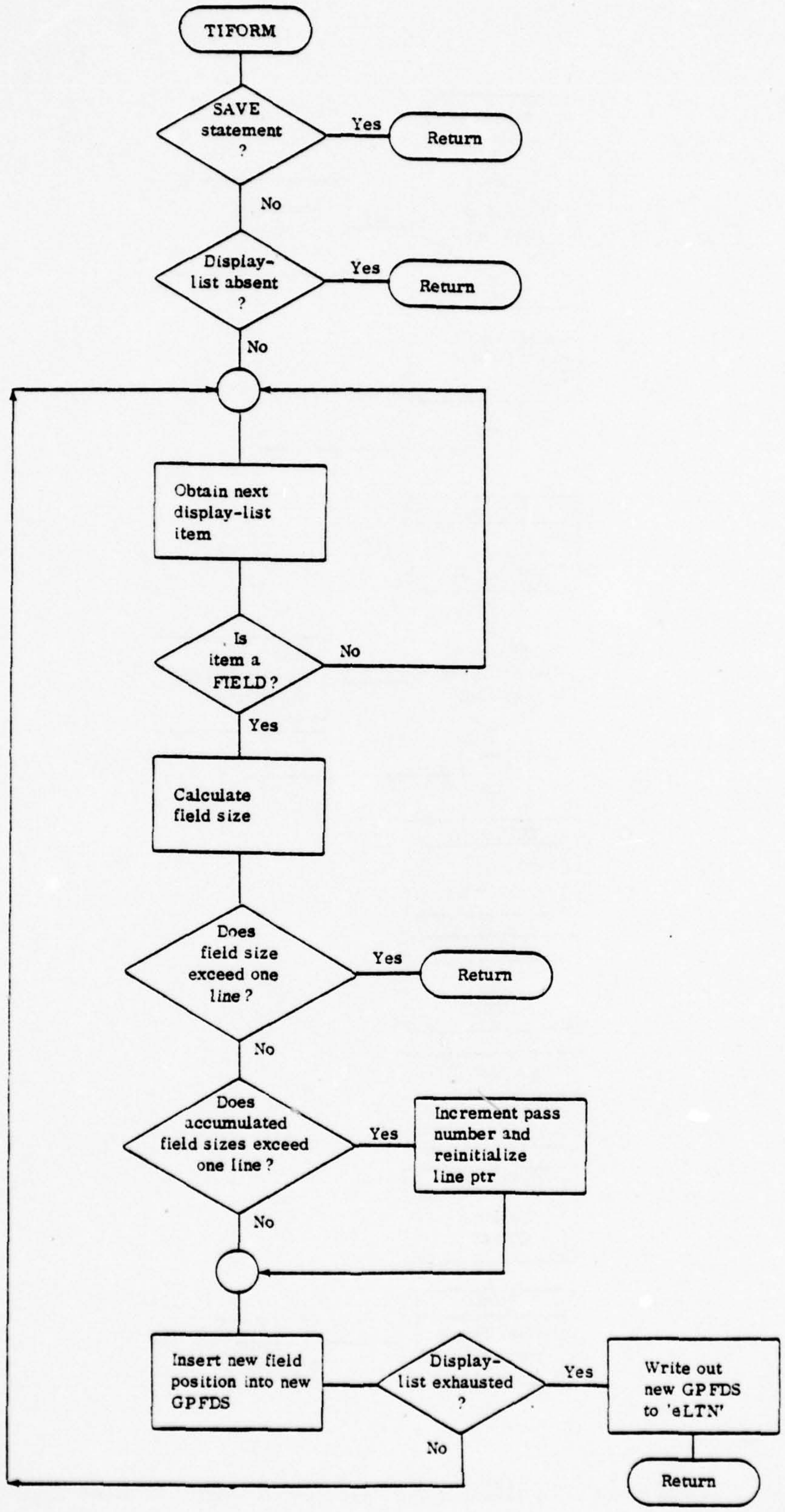

TIFORM
SAVE statement ?
Yes
Return
No
Display-list absent ?
Yes
Return
No
Obtain next display-list item
Is item a FIELD ?
No
Yes
Calculate field size
Does field size exceed one line ?
Yes
Return
No
Does accumulated field sizes exceed one line ?
Yes
Increment pass number and reinitialize line ptr
No
Insert new field position into new GPFDS
Display-list exhausted ?
Yes
Write out new GPFDS to 'eLTN'
Return
No

Figure 46. TSIGB Process Data Flow (Sheet 3 of 7)

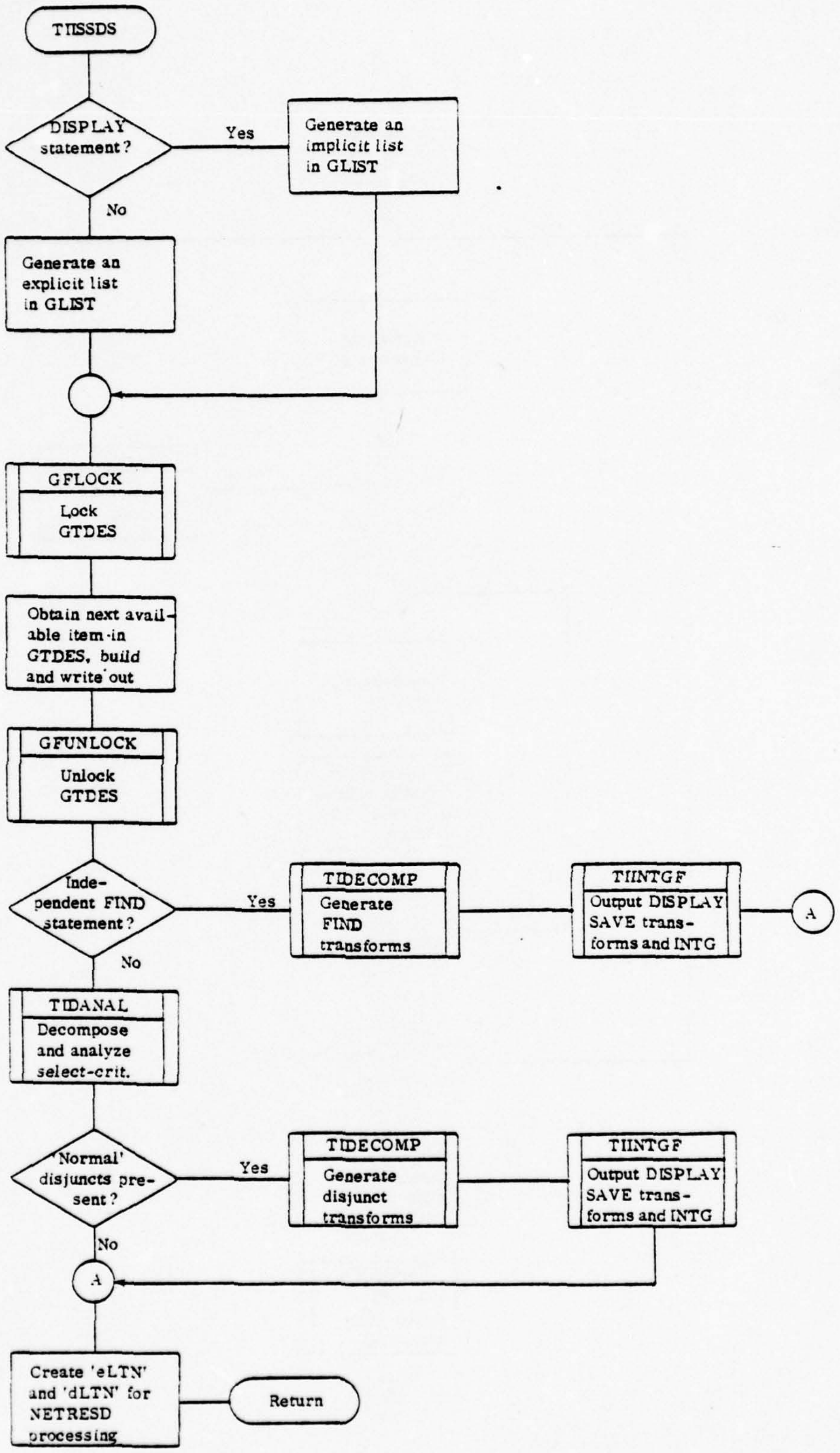

TIISSDS
DISPLAY statement?
Yes
Generate an implicit list in GLIST
No
Generate an explicit list in GLIST
GFLOCK
Lock GTDES
Obtain next available item in GTDES, build and write out
GFUNLOCK
Unlock GTDES
Independent FIND statement?
Yes
TIDECOMP
Generate FIND transforms
TIINTGF
Output DISPLAY SAVE transforms and INTG
A
No
TIDANAL
Decompose and analyze select-crit.
'Normal' disjuncts present?
Yes
TIDECOMP
Generate disjunct transforms
TIINTGF
Output DISPLAY SAVE transforms and INTG
No
A
Create 'eLTN' and 'dLTN' for NETRESD processing
Return

Figure 46. TSIGB Process Data Flow (Sheet 4 of 7)

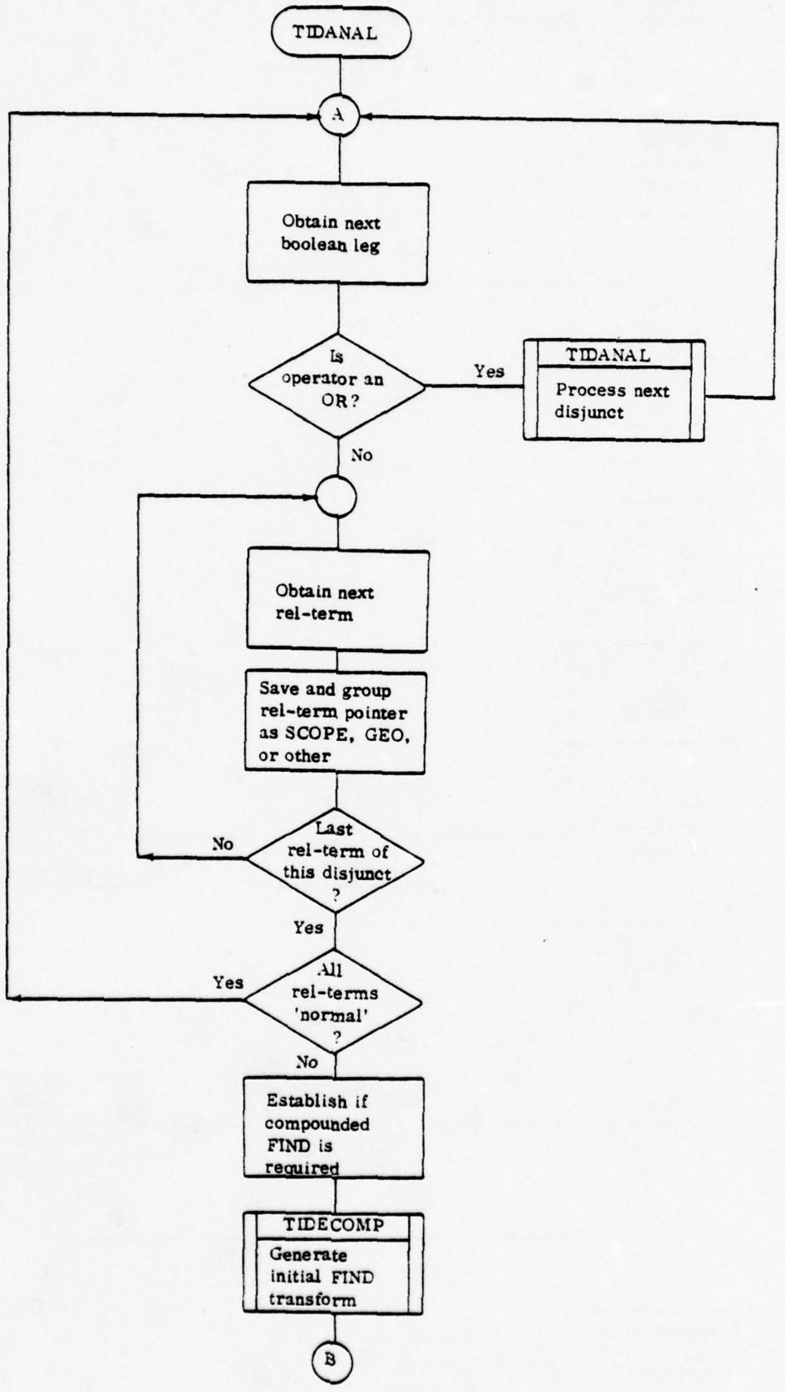

TIDANAL
A
Obtain next boolean leg
Is operator an OR?
Yes
TIDANAL
Process next disjunct
No
Obtain next rel-term
Save and group rel-term pointer as SCOPE, GEO, or other
Last rel-term of this disjunct ?
No
Yes
All rel-terms 'normal' ?
Yes
No
Establish if compounded FIND is required
TIDECOMP
Generate initial FIND transform
B

Figure 46. TSIGB Process Data Flow (Sheet 5 of 7)

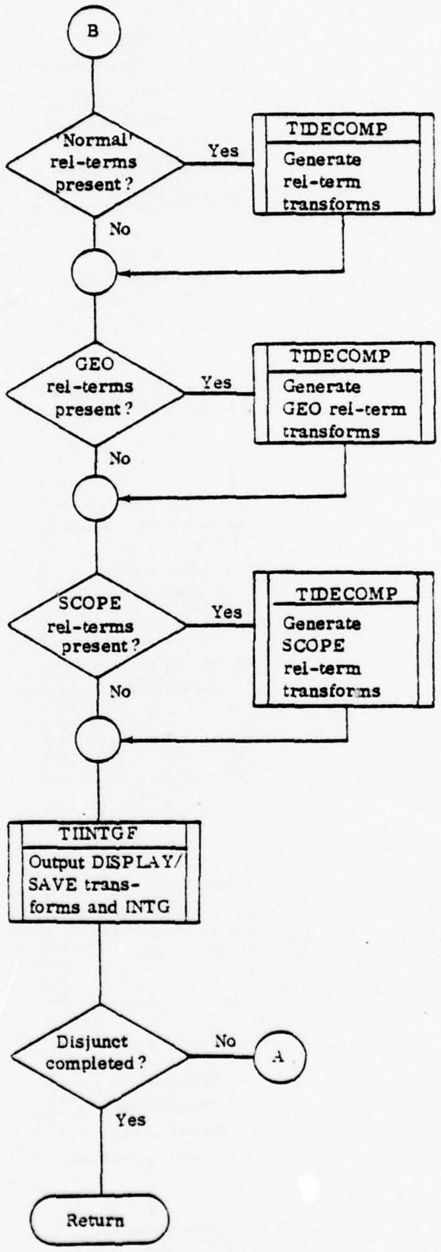

B
'Normal' rel-terms present?
Yes
No
TIDECOMP
Generate rel-term transforms
GEO rel-terms present?
Yes
No
TIDECOMP
Generate GEO rel-term transforms
SCOPE rel-terms present?
Yes
No
TIDECOMP
Generate SCOPE rel-term transforms
TIINTGF
Output DISPLAY/ SAVE trans-forms and INTG
Disjunct completed?
No
A
Yes
Return

Figure 46. TSIGB Process Data Flow (Sheet 6 of 7)

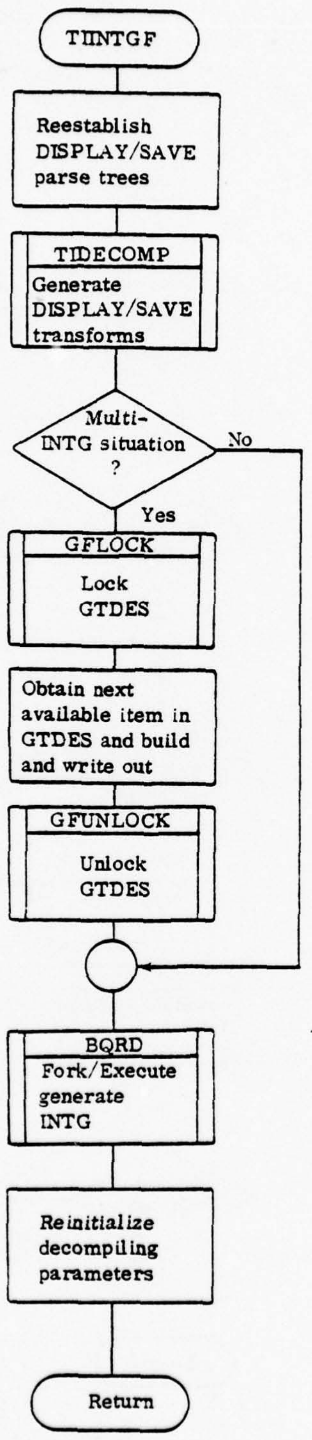

TIINTGF
Reestablish DISPLAY/SAVE parse trees
TIDECOMP
Generate DISPLAY/SAVE transforms
Multi-INTG situation ?
No
Yes
GFLOCK
Lock GTDES
Obtain next available item in GTDES and build and write out
GFUNLOCK
Unlock GTDES
BQRD
Fork/Execute generate INTG
Reinitialize decompiling parameters
Return

Figure 46. TSIGB Process Data Flow (Sheet 7 of 7)

TSIGI (Target System Input Generator: Interactive)

The TSIGI process is the ADAPT I process which controls the conversion of UDL statements into appropriate interactive target system transformation strings. This process provides both of the functions performed by TSIGB and NETRES processes for batch interrogations. The primary difference between interactive and batch interrogations is that a user is essentially connected to an interactive host system when referencing and receiving data from some file whereas, in a batch system, the acts of interrogation and the reception of subsequent responses are two separate operations which may sometimes be separated by a lengthy time interval.

GLOBAL DATA USAGE – TSIGI utilizes the following global data:

| | | |
|---|---|---|
| GLABL | – | label names and descriptions. |
| GLIST | – | list names and descriptions. |
| GTREE | – | parse tree. |
| GTDAT | – | parse tree data. |
| GFILD | – | file descriptions. |
| GTFLN | – | transformation file names. |

LOCAL DATA USAGE – All three local data structures, TITRG, TIPRM, and TIACT, which are utilized by TIDECOMP, contain target system specific information with respect to query language constructs.

| | | |
|---|---|---|
| TIPRM | – | used by TIDECOMP to point to appropriate transformation string constructs in TITRG and to appropriate action functions in TIACT. TIPRM is indexed by UDL tokens contained in GTREE. |

| | |
|---|---|
| TITRG – | contains actual transformation string constructs which represent the UDL token in the target system query language. TITRG is indexed by TIPRM. |
| TIACT – | contains switch pointers to appropriate action functions and parse tree leg indicators for function arguments. TIACT is pointed to by TIPRM and its ordered access is activated by escape characters in the transformation string constructs in TITRG. |

GENERAL PROCESS FLOW – Under ADAPT I the TSIGI process is actually composed of two processes: TSIG3 for SOLIS, and TSIG4 for QLP. However, since both hosts are interactive, the overall data flow of both processes is quite similar and is described here as a unit. Differences between them exist as host specific functions described below.

TSIGI is called either by the BEN process after it has successfully converted the FIND statement selection criteria to DNF, or by the VALIDATE process after encountering a DISPLAY statement. In both cases, either the FIND or DISPLAY statement must reference a file that resides on an interactive host. TSIGI first opens and reads in the user's label file, GLABL, which represents the user's current logon status for this ADAPT I session. Then the initial blocks of the parse tree data files, GTREE and GTDAT, are read in and TSIGI determines whether the statement is a FIND statement or a DISPLAY statement. If the statement is a DISPLAY statement, TSIGI calls the host specific function (TISOLDS for SOLIS, TIQLPDS for QLP) to process the DISPLAY and then exits. If the statement is a FIND statement, TSIGI updates the user's label file, GLABL, inserting the new label (the label prefixing the FIND statement). If more than six labels are in GLABL, the oldest label is removed including all other labels which may reference it (i. e., labels on dependent FIND statements). Then TSIGI calls the host specific function which will perform the upper level FIND statement processing (see TISOLF for SOLIS and TIQLPF for QLP). This logic path initiates the required host connection and generates the appropriate transforms via the decompiler, TIDECOMP.

After sending the transformed query to the proper host, the resulting hit count response is analyzed. This hit count is output to the user's terminal and is also stored in the user's label file, GLABL, accompanying the label of the FIND statement. This hit count is used by the VALIDATE process to ensure that the user does not attempt to display record data from a set of records exceeding the allowable number. Following the return from the host specific function, TSIGI then writes out the user's label file, GLABL, and exits back to the EXEC Process. (Note: although BEN called TSIGI, it did not wait for its completion, hence TSIGI returns to EXEC which indirectly called it.) If the host specific function returned with a negative status, indicating one of many abnormal situations, TSIGI removes the FIND statement label from GLABL prior to writing it out. Figure 47 provides more information on the TSIGI data flow.

## MAJOR FUNCTION DESCRIPTIONS

TISOLF (SOLIS FIND Statement Processor) – This function is responsible for performing upper level processing of UDL FIND statements which must be transformed to SOLIS constructs. First, TISOLF must call TILOGON to establish the initial connection (if not already connected). If TILOGON returns with a negative value, either the host was not available or the user was not properly authenticated by the SOLIS host. In either case, TISOLF returns with a negative one. If the user is already connected to SOLIS or the connection sequence was successful, TISOLF then calls TINITQU to initiate the query to SOLIS. If TINITQU returns with a negative value, then the appropriate volumes were not available for this given UDL file (available product volumes on a given day are variable) and TISOLF returns with a negative one. If TINITQU returns successfully, TISOLF then examines the resulting QM screen for the hit count from the query. This hit count is then output to the user's terminal. TISOLF also returns this resulting hit count to TSIGI.

TISOLDS (SOLIS DISPLAY Statement Processor) – This function is responsible for analyzing a display-list and determining the proper SOLIS output mode for DISPLAY processing. First, TISOLDS analyzes the display-list, if present. If the display-list is not present, the user is requesting all data from the selected records. If the display-list is present, TISOLDS searches for a reference to the TEXT portion of the record. If it is not found, the user is only viewing one or more fields derivable from the SOLIS Preamble A0 option. If the TEXT reference is found, the user is interested in viewing the entire record text (i. e., entire text of the selected messages). Finally, based on this analysis, a TSOT mode is established as follows:

a. TSOT mode of 0 implies output Preamble A0 only.
b. TSOT mode of 1 implies output both text and Preamble A0.
c. TSOT mode of 2 implies output text only.

TISOLDS now generates an implicit entry into GLIST and then calls the appropriate TSOT process. TSOT builds the appropriate GRMAP/GRDAT pairs and calls the DISPLAY process when the first record is generated. Upon completion of this process, TSOT returns, and TISOLDS deletes the implicit list entry from GLIST and returns to the calling function.

TINITQU (Initiate Query to SOLIS) – This function is responsible for establishing the correct database for SOLIS retrieval (via a call to TIFILE) and outputting the FIND statement transform to SOLIS (via a call to TIDECOMP). First, TINITQU calls TIFILE to establish the correct database and the proper volumes in SOLIS. Before returning, TIFILE also establishes the SOLIS free query screen as the current screen. Then TINITQU calls the decompiler, TIDECOMP, to actually generate the SOLIS transforms for the input FIND statement. Finally, TINITQU establishes the SOLIS QM screen as the current screen. This screen is used by TISOLF to extract the hit count of the query.

TIFILE (SOLIS File/Volume Control Processor) – This function is responsible for controlling all SOLIS screen path processing based on the UDL file currently being referenced. First, TIFILE must determine the TDL file name mapping for this UDL file. This datum is contained in GTFLN. Then, TIFILE determines if the present file was also the last file referenced by this user. If it is, then only a free screen must be established for the next query. If it is not, TIFILE checks to see if this statement initiated the connection to the SOLIS host (i.e., the first FIND statement for this host), and if so, must check to see if the desired file is SIRE. If it is, the product volume suppression screen is bypassed and TIFILE scans for a query screen. Upon receipt of the query screen, an RS command is output to SOLIS specifying that retrieval from SIRE is desired. If the desired file is not SIRE, the volume suppression screen is processed by TIVOLSUP, and a free screen is established for the next query. If this is not the first statement for this connection to SOLIS and the desired file is not SIRE, TIFILE checks to see if the previous FIND statement referenced the SIRE database. If it did not, TIFILE is in the correct database and only has to reestablish the volume suppression screen required by the current PRODUCT file. If the previous reference was to the SIRE database, an RP command must be sent to SOLIS in order to change databases. Following this, the volume suppression screen is processed by TIVOLSUP and a free screen is established for the next query.

TIVOLSUP (Volume Suppression Processor) – This function is responsible for processing a volume suppression screen received from the SOLIS host. First, TIVOLSUP searches the volume screen until it reaches the individual volume specifications (i.e., single alphabetic characters representing certain date spans of available database volumes). Then for each available volume found on the screen, TIVOLSUP searches the allowable volumes for this given UDL file. If the volume is not allowable, TIVOLSUP inserts an X character into the appropriate position of a fabricated SOLIS screen. If the volume is allowable, a space character is output. Also, if the UDL file is the SIRE database or specifies the PRODUCT database and all volumes are allowed, TIVOLSUP will select all volumes. If TIVOLSUP discovers that the UDL file does not map onto at least one available volume, the appropriate diagnostic is output to the user and TIVOLSUP returns with a negative one. Otherwise, it outputs the completed volume suppression screen to the SOLIS host. Note, all volumes are selected for SIRE in order to bypass this screen. The screen cannot be ignored, it must be sent in a valid form to SOLIS in order to continue.

TIQLPF (QLP FIND Statement Processor) – This function is responsible for upper level processing of a FIND statement which is to be transformed to QLP constructs. First, this function calls TILOGON to initiate the connection to the QLP host. Possibly the user is already connected to this host and TILOGON returns immediately with a successful connection status. If the user is not connected to the host, then TILOGON must initiate the connection. If TILOGON does not return with a successful status, TIQLPF returns to TSIGI with a negative one. If the connection status is successful (a zero) then TIQLPF calls the decompiler, TIDECOMP, to generate the appropriate transforms to QLP. Then TIQLPF calls TILOGON to read in the host response. Upon completion of the read, TIQLPF searches the response for the hit count which is then output to the user's terminal, and TIQLPF returns to the calling function.

TIQLPDS (QLP DISPLAY Statement Processor) – This function is responsible for performing upper level processing of DISPLAY statements going to the QLP host. This function first calls TILOGON to ensure that the user is still connected to this host. If the user is not connected, the connection must be reestablished. If TILOGON returns a negative status, either the connection was not successful or the user was not authenticated properly. For this case, TIQLPDS returns with a negative value. Assuming successful connection, TIQLPDS generates an implicit list entry in

GLIST and calls the decompiler, TIDECOMP, to generate the proper constructs for QLP. Following the output of the transforms to the QLP, the response is normalized and written into file cOOO, and TIQLPDS calls the appropriate TSOT process in order to generate the GRMAP/GRDAT pairs (internal records) for this host response. TSOT is responsible for calling the DISPLAY process when the first record is generated. The DISPLAY process will output directly to the user's terminal. Upon completion of this sequence TSOT exits, and TIQLPDS removes the implicit list entry from GLIST and returns to TSIGI.

TILOGON (Host Logon) – This function is responsible for calling the TAS Interactive Query Interface (IQI) process for establishing a connection to an interactive host on the COINS II network. TILOGON is also responsible for reading host data from an existing connection. TILOGON first determines whether this is a logon request or a read request. If it is a logon request, it checks to see if this user is already connected to the desired host. If the desired host is already connected to ADAPT I, TILOGON removes any residue data on the pipe that may have been left over from a previous interaction (i. e., the user can terminate a DISPLAY output anytime during its operation). Once this is accomplished, TILOGON returns. If the user is not presently connected to the indicated host, TILOGON must first determine if the user is connected to another interactive host and if he is, terminate this connection. Currently under ADAPT I and TAS, a given user/terminal set can only be connected to one interactive host at any given time. The connection is terminated by sending a signal 7 to IQI. Now TILOGON performs a fork/execute sequence to call IQI for the new connection, passing it the host ID, appropriate file descriptors of two pipes, and TSIGI's process ID. TILOGON waits for the status by reading on the pipe. IQI will return one word indicating that the connection and transparent user logon was successful (a zero), the connection was not successful (a negative one), or the connection was successful but the user logon was not (a negative two). If an unsuccessful indication is received, TILOGON returns with a negative one. Otherwise, TILOGON updates the user's GLABL item zero reflecting the new host connection and returns with a zero. If the calling function requested a read, TILOGON reads from the pipe until an ETX character is encountered. It is assumed that all data coming from the host will be terminated by an ETX character. Upon encountering an ETX, TILOGON returns with a count reflecting the number of characters read.

TIDECOMP (Decompiler) – The decompiler, TIDECOMP, is responsible for the generation of transformation strings for the various target systems across the network. TIDECOMP, when called, is passed a pointer into a parse tree, GTREE. Usually this pointer references the statement/command token at the head of the tree, although this is not a requirement. TIDECOMP uses the primitive UDL token as an index into local table TIPRM. TIPRM points into two other tables: transformation string table TITRG and action table TIACT. The pointer to TITRG points to a set of characters representing the UDL token for that target system. The pointer to TIACT points to a set of action indicators that are required to satisfy the transformation of the UDL token. The various actions are triggered by TIDECOMP control characters in TITRG. From this point on, TITRG controls the actions of TIDECOMP.

Each character in TITRG (starting with the character pointed to initially by TIPRM) is analyzed to see if it is one of three TIDECOMP control characters: "\0", "\\", and "∧". The null character "\0" indicates the end of the transformation string for a UDL token, the backslash character "\\" indicates the execution of an action function, and the character "∧" indicates a list token. If the character is not a control character, it is output as a transformation string character. The null character causes

TIDECOMP to return. The backslash character causes TIDECOMP to perform the next action contained in TIACT (starting initially with the action pointed to by TIPRM). TIACT also specifies an optional relative parse tree leg pointer for input to the action. The majority of action functions utilized by TIDECOMP are general in nature and hence are used for most target system transformations. See appendix C for a list of these action functions. The character "∧" tells TIDECOMP that a list token is present. TIDECOMP uses the next two characters in TITRG to determine how many list-separator characters are required and the position in TITRG at which to continue, following completion of the list. The indicated list-separator characters follow the control characters. The actual decompiling tables used in ADAPT I for the four hosts are presented in appendix C.

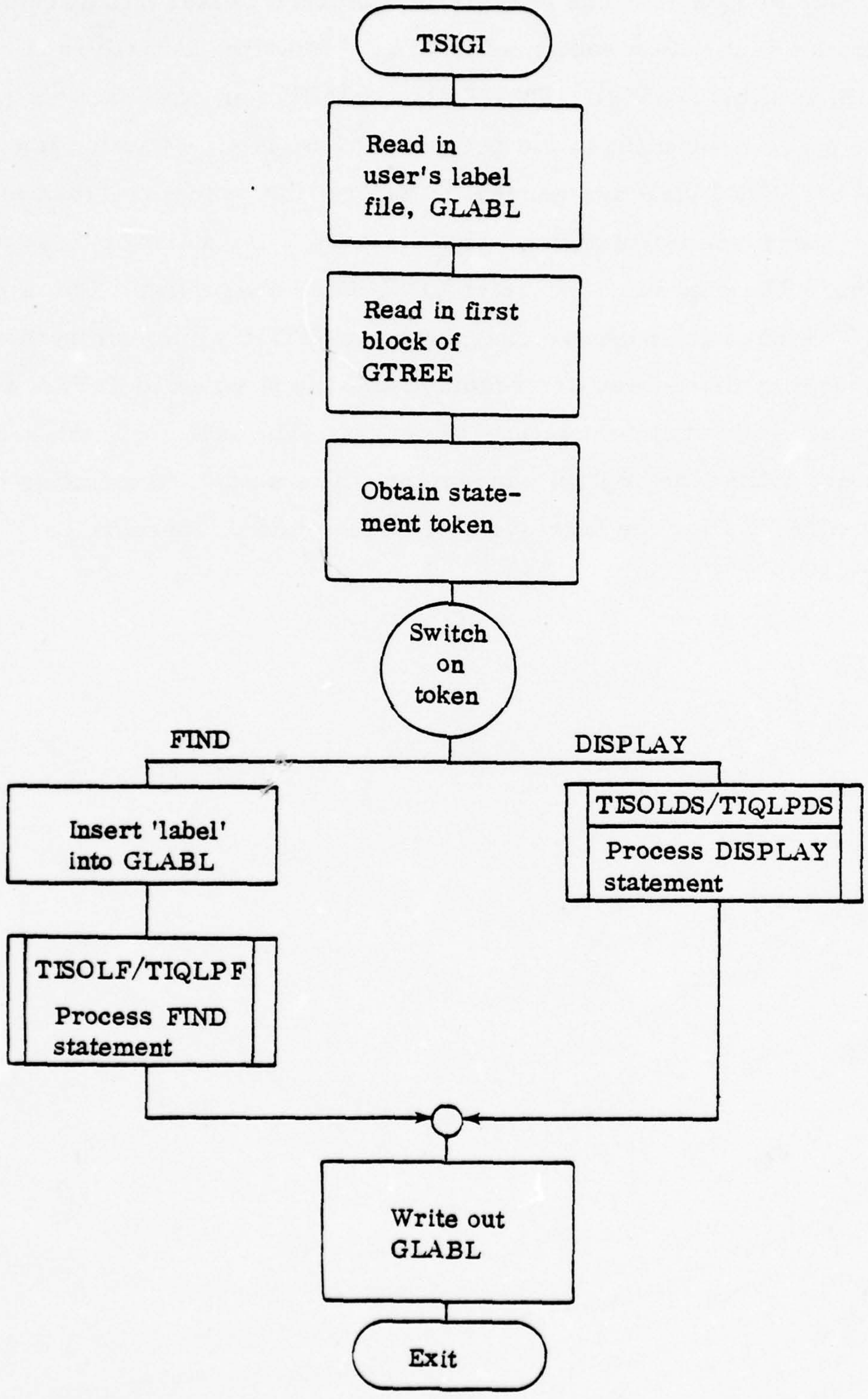


Figure 47. TSIGI Process Data Flow (Sheet 1 of 10)

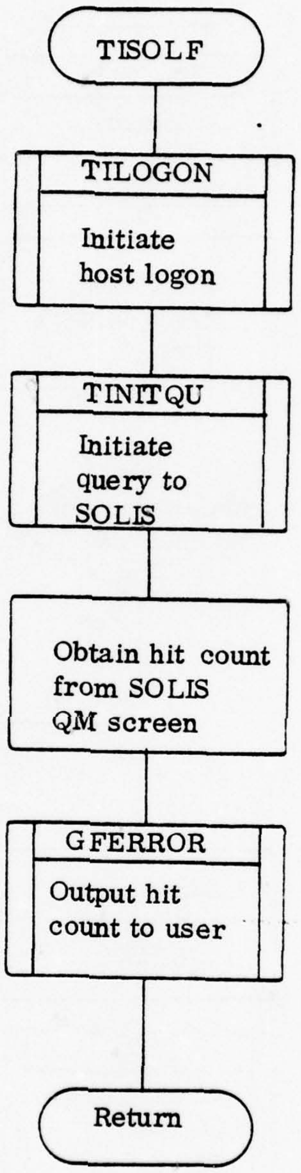


Figure 47. TSIGI Process Data Flow (Sheet 2 of 10)

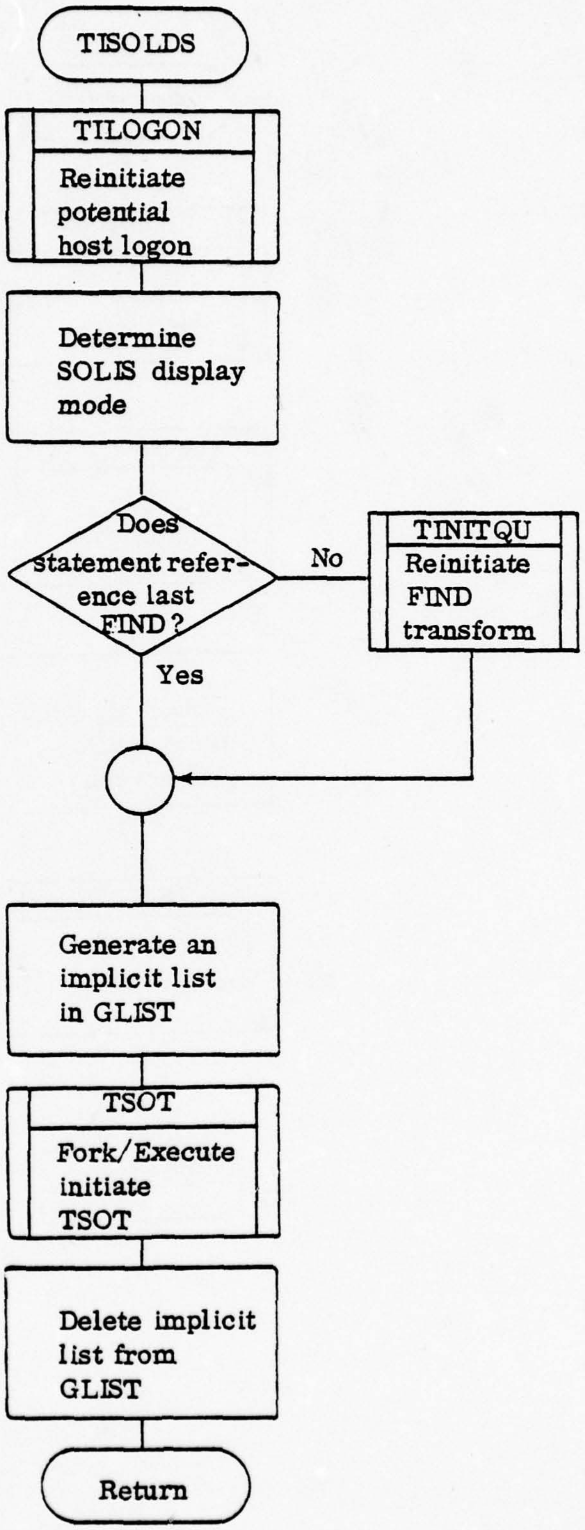

TISOLDS
TILOGON
Reinitiate potential host logon
Determine SOLIS display mode
Does statement reference last FIND?
No
TINITQU
Reinitiate FIND transform
Yes
Generate an implicit list in GLIST
TSOT
Fork/Execute initiate TSOT
Delete implicit list from GLIST
Return

Figure 47. TSIGI Process Data Flow (Sheet 3 of 10)

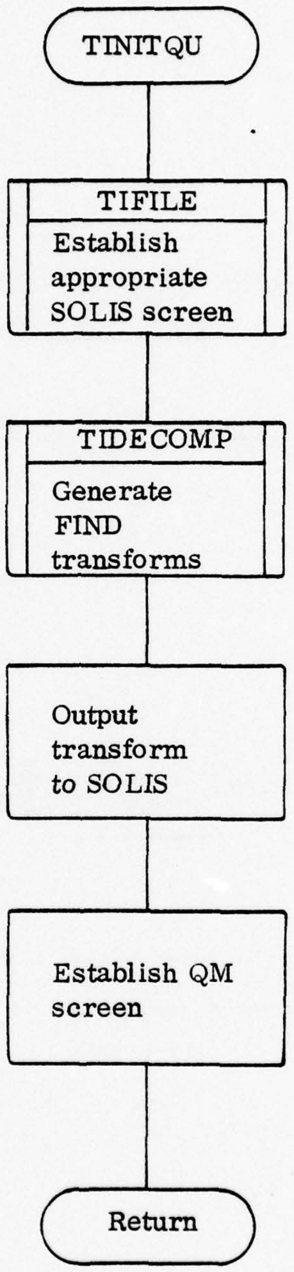


Figure 47. TSIGI Process Data Flow (Sheet 4 of 10)

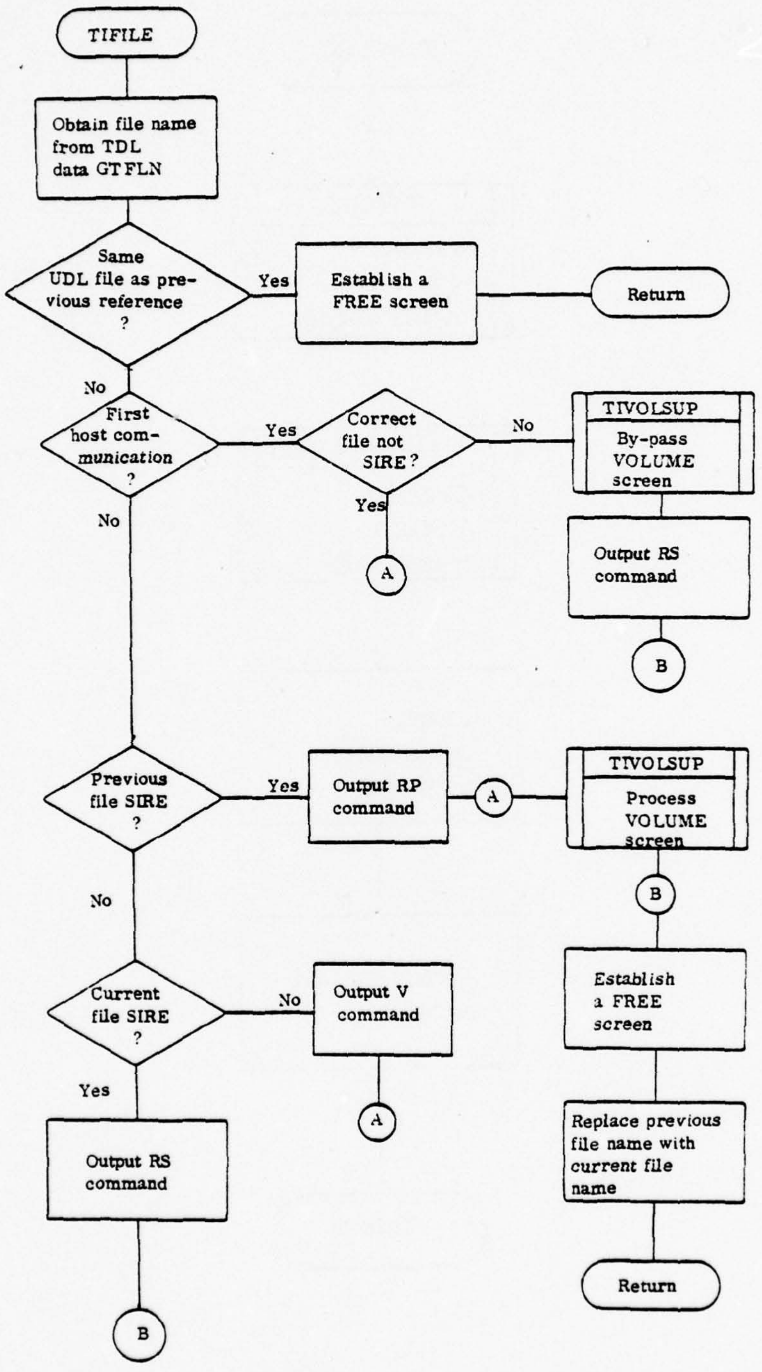


Figure 47. TSIGI Process Data Flow (Sheet 5 of 10)

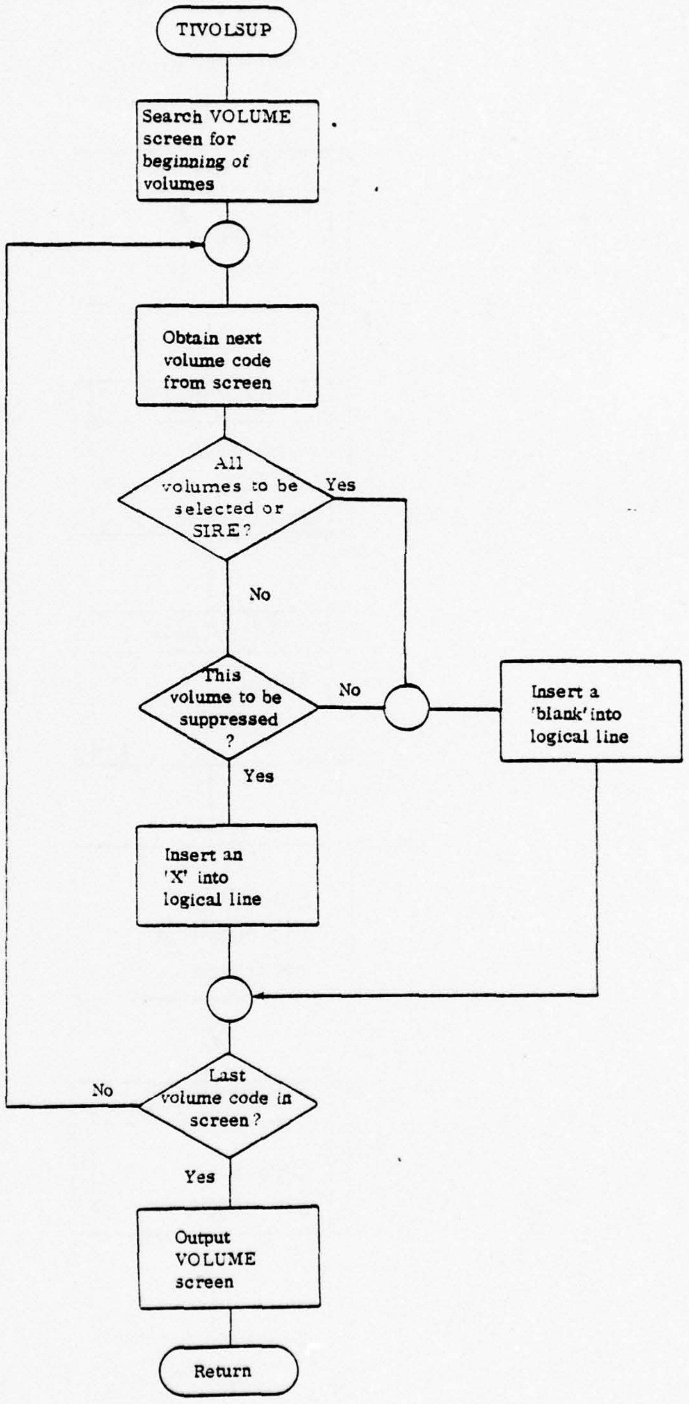

TIVOLSUP
Search VOLUME screen for beginning of volumes
Obtain next volume code from screen
All volumes to be selected or SIRE?
Yes
No
This volume to be suppressed ?
No
Yes
Insert a 'blank' into logical line
Insert an 'X' into logical line
Last volume code in screen ?
No
Yes
Output VOLUME screen
Return

Figure 47. TSIGI Process Data Flow (Sheet 6 of 10)

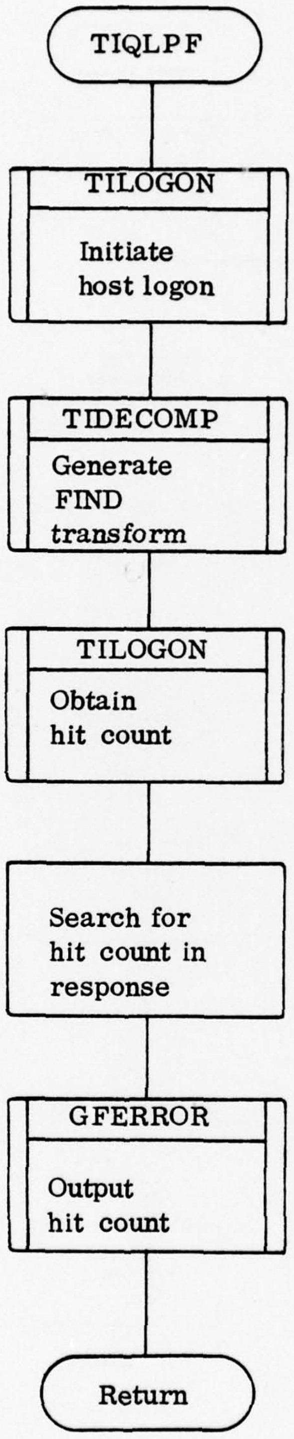


Figure 47. TSIGI Process Data Flow (Sheet 7 of 10)

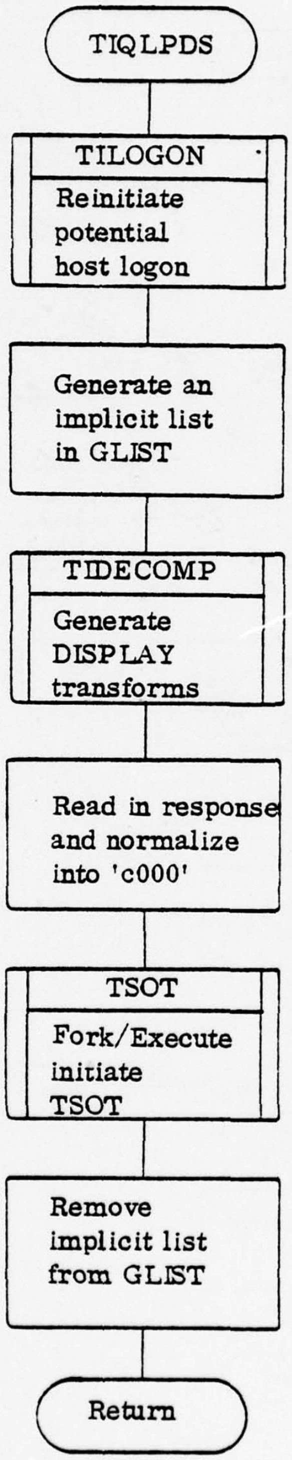

TIQLPDS
TILOGON
Reinitiate potential host logon
Generate an implicit list in GLIST
TIDECOMP
Generate DISPLAY transforms
Read in response and normalize into 'c000'
TSOT
Fork/Execute initiate TSOT
Remove implicit list from GLIST
Return

Figure 47. TSIGI Process Data Flow (Sheet 8 of 10)

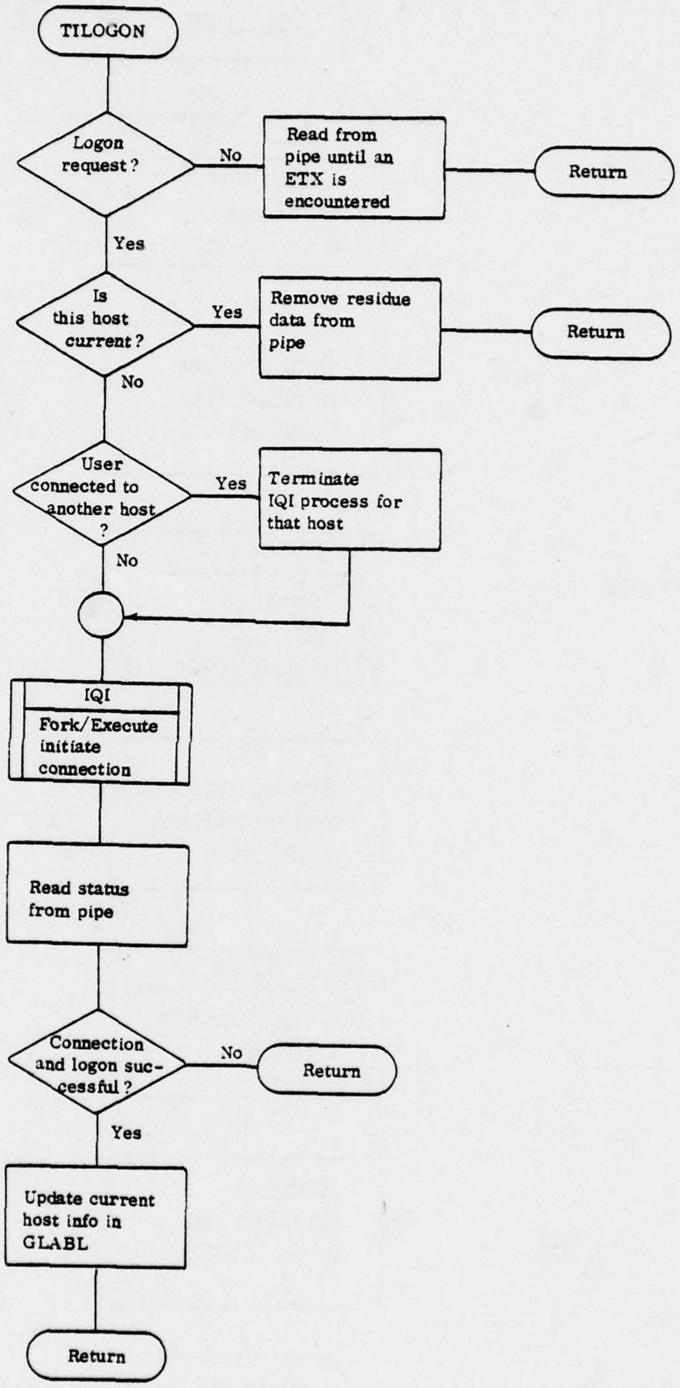

TILOGON
Logon request?
No
Read from pipe until an ETX is encountered
Return
Yes
Is this host current?
Yes
Remove residue data from pipe
Return
No
User connected to another host ?
Yes
Terminate IQI process for that host
No
IQI
Fork/Execute initiate connection
Read status from pipe
Connection and logon successful ?
No
Return
Yes
Update current host info in GLABL
Return

Figure 47. TSIGI Process Data Flow (Sheet 9 of 10)

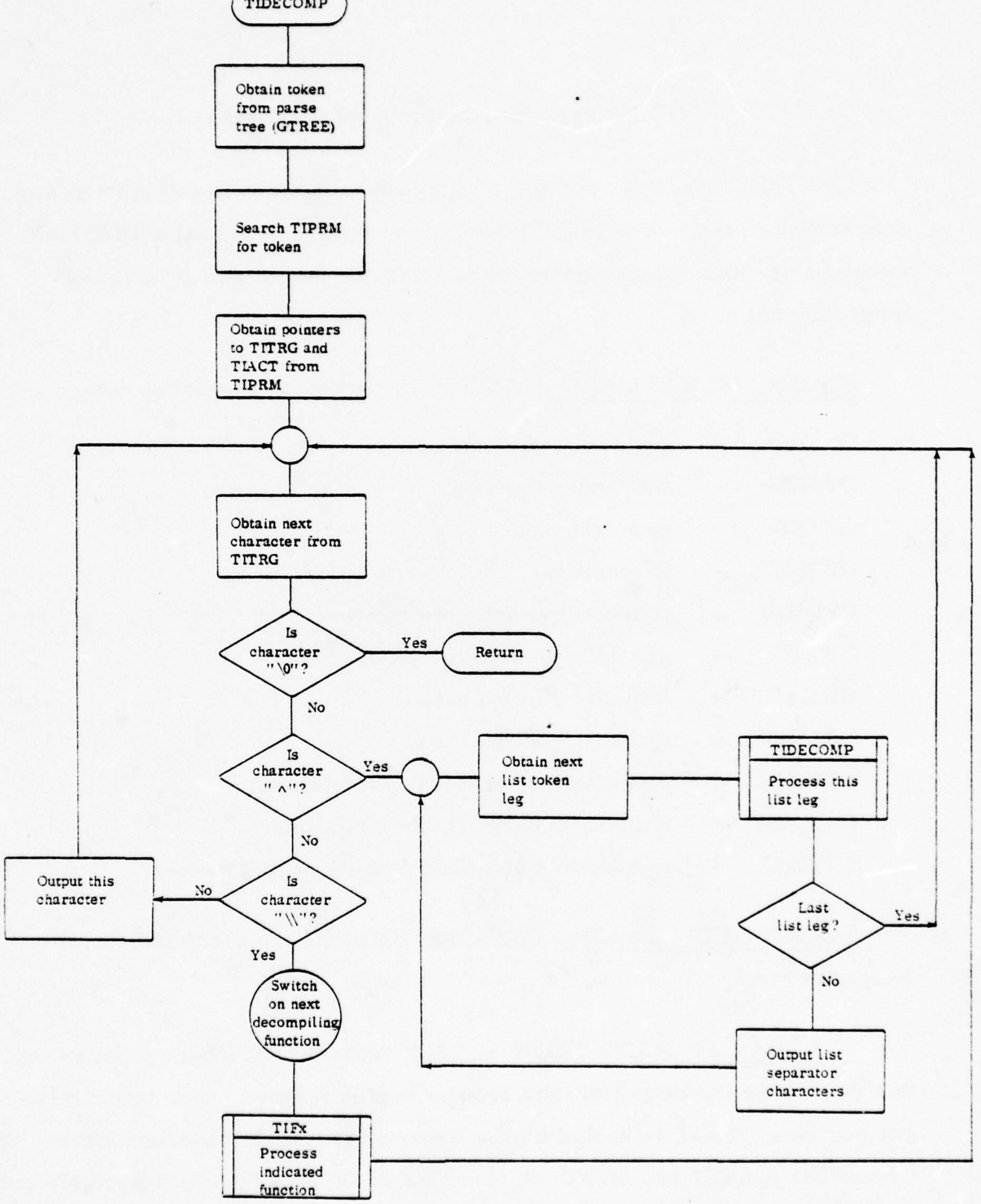

TIDECOMP
Obtain token from parse tree (GTREE)
Search TIPRM for token
Obtain pointers to TITRG and TIACT from TIPRM
Obtain next character from TITRG
Is character "\0"?
Yes
Return
No
Is character "^"?
Yes
Obtain next list token leg
TIDECOMP
Process this list leg
Last list leg?
Yes
No
Output list separator characters
No
Is character "\\"?
No
Output this character
Yes
Switch on next decompiling function
TIFx
Process indicated function

Figure 47. TSIGI Process Data Flow (Sheet 10 of 10)

## TSOT (Target System Output Translator)

The TSOT process translates output data from a target system and converts the data into ADAPT internal records. TSOT calls DISPLAY to output the data interactively to the user for data from interactive target systems.

GLOBAL DATA USAGE – TSOT uses the following global data:

| | | |
|---|---|---|
| GADES | – | aggregate descriptions. |
| GFDES | – | field descriptions. |
| GFILD | – | file descriptions. |
| GFILN | – | file names. |
| GMPHD | – | internal record map header. |
| GPFDS | – | position field descriptions. |
| GRDAT | – | internal record data. |
| GRMAP | – | internal record map. |
| GTFDS | – | transformation field descriptions. |
| GTFLD | – | transformation file descriptions. |
| GTNAM | – | transformation field and aggregate names. |

LOCAL DATA USAGE – TSOT uses TOBUFF, input buffer for response strings.

GENERAL PROCESS FLOW – TSOT is activated whenever a user has requested the output of data from a target system. For interactive target systems, TSOT is called by the appropriate Target System Input Generator (TSIGI) and TSOT, in turn, calls DISPLAY to interactively output the data as it is being built. For batch target systems, TSOT is called by the appropriate Network Response Processor (NETRES), and after building the internal records, TSOT terminates. As the target systems have rather different response data, the TSOT processes will be written up separately for the sake of clarity. In the following discussions and flow charts, they will be referred to as: TSOTD (TSOT for DIAOLS, batch), TSOTQ (TSOT for QLP, interactive), TSOTR (TSOT for RYETIP, batch), and TSOTS (TSOT for SOLIS, interactive).

## TSOTD (TSOT for DIAOLS)

GENERAL PROCESS FLOW – TSOTD picks up the GRMAP and GRDAT file descriptors passed by NETRESD. It then reads the following files to be used in translating the response data and converting it into internal records: GTFLD, GFILN, GFILD, GTFDS, GTNAM, GFDES, and GADES. Function TOFNAM is called to process the DIAOLS name recognition data. After the response data has been processed, TSOTD writes a null Internal Record Map Header (GMPHD) and exits. See Figure 48 for data flow.

### MAJOR FUNCTION DESCRIPTIONS

TOFNAM (Process Field Name Recognition Data) – TOFNAM searches for a field name, reading response strings as needed. If it recognizes end-of-data, it stores any occurrence nodes which have been saved in a stack into GRMAP, then writes out the record map header (GMPHD), record map (GRMAP), and final record data block (GRDAT) and returns. If TOFNAM has found a field name, it finds a match in the transformation field data (GTFDS) and saves the field's index and size. TOFNAM determines whether this field starts a new record and, if so, stores occurrence nodes in GRMAP and writes out the files as in the preceding. If the field has no data value, TOFNAM proceeds to the next field. If the field is not in the basic data set, TOFNAM saves the aggregate indexes of the field's parent aggregate and its superordinate aggregates, if any. If this is the first field encountered for its aggregate, an occurrence node is started in a stack and a dataset node is stored in GRMAP. If this field starts a new occurrence for its aggregate, the existing occurrence node in the stack is updated and a dataset node is stored in GRMAP. Pointers to the dataset nodes are stored in the occurrence node in the stack with the occurrence count. For all fields which have a data value, TOFNAM calculates an index into GRDAT. If the current GRDAT block is full, it is written out and a new block is started. TOFNAM reads the field's value(s) and stores the data in GRDAT. A pointer and occurrence count are stored in GRMAP for each multivalued and variable-length field. After processing this field, TOFNAM goes on to the next field.

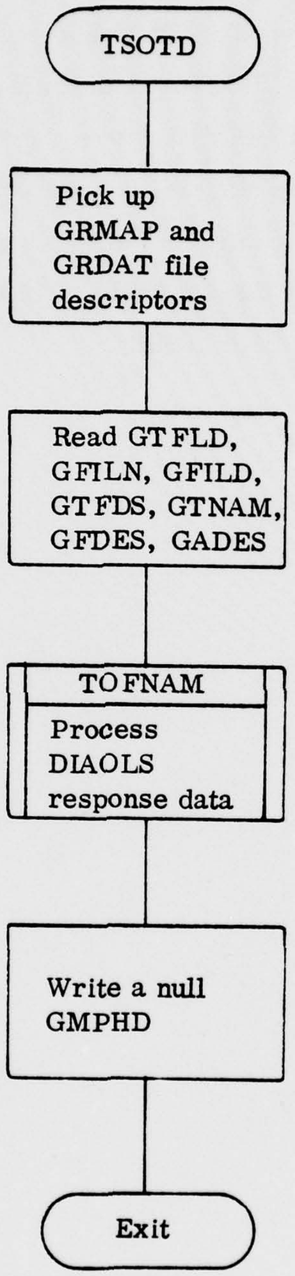

TSOTD
Pick up GRMAP and GRDAT file descriptors
Read GTFLD, GFILN, GFILD, GTFDS, GTNAM, GFDES, GADES
TOFNAM
Process DIAOLS response data
Write a null GMPHD
Exit

Figure 48. TSOTD Process Data Flow (Sheet 1 of 3).

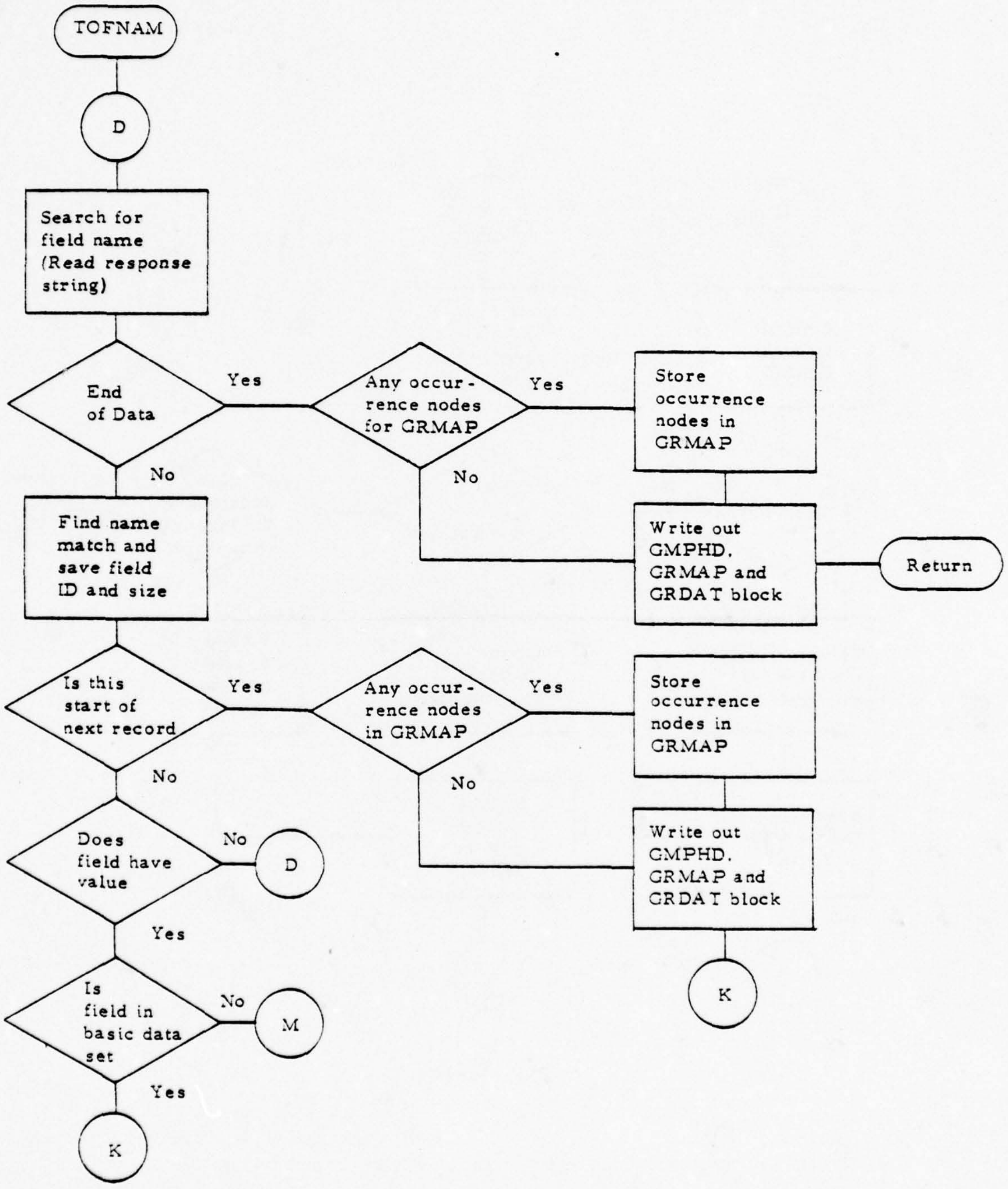

TOFNAM
D
Search for field name (Read response string)
End of Data
Yes
No
Any occurrence nodes for GRMAP
Yes
No
Store occurrence nodes in GRMAP
Write out GMPHD, GRMAP and GRDAT block
Return
Find name match and save field ID and size
Is this start of next record
Yes
No
Any occurrence nodes in GRMAP
Yes
No
Store occurrence nodes in GRMAP
Write out GMPHD, GRMAP and GRDAT block
K
Does field have value
No
D
Yes
Is field in basic data set
No
M
Yes
K

Figure 48. TSOTD Process Data Flow (Sheet 2 of 3).

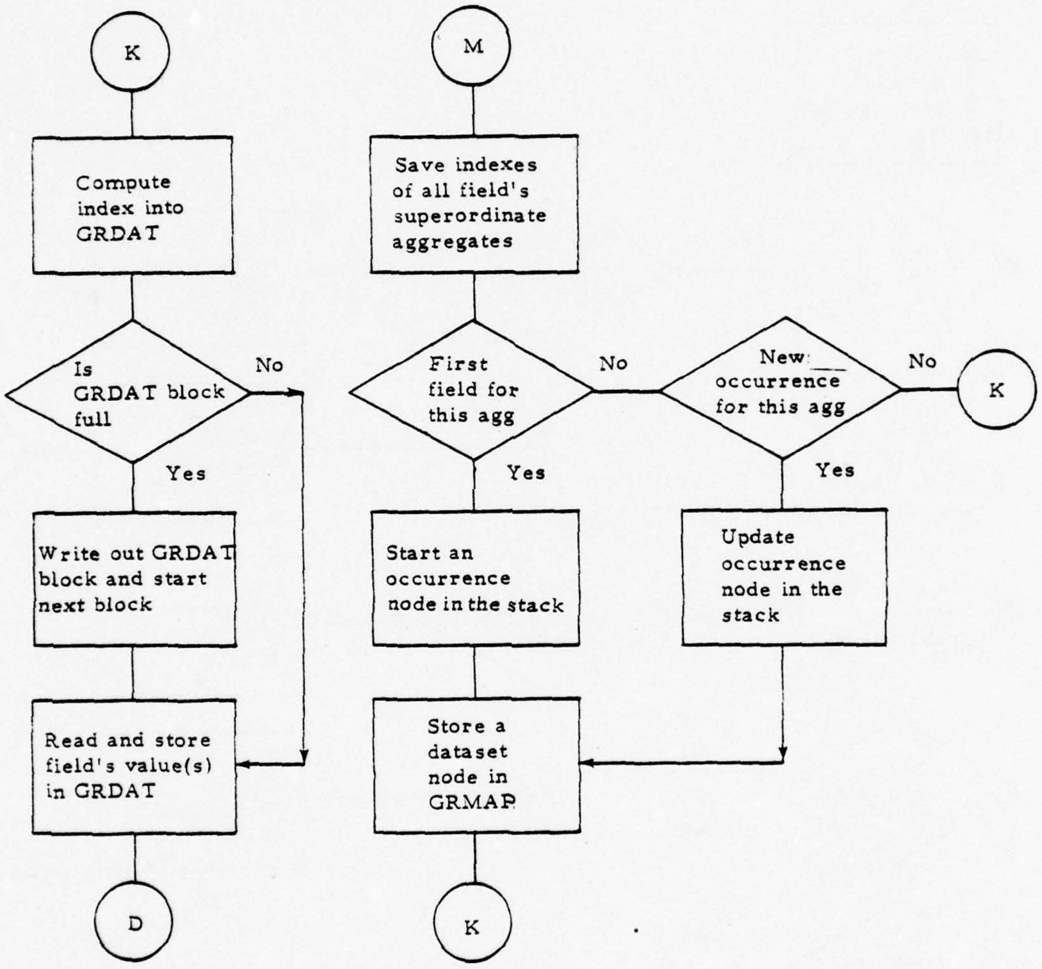

K
Compute index into GRDAT
Is GRDAT block full
No
Yes
Write out GRDAT block and start next block
Read and store field's value(s) in GRDAT
D
M
Save indexes of all field's superordinate aggregates
First field for this agg
No
Yes
Start an occurrence node in the stack
Store a dataset node in GRMAP
K
New occurrence for this agg
No
Yes
K
Update occurrence node in the stack

Figure 48. TSOTD Process Data Flow (Sheet 3 of 3).

## TSOTQ (TSOT for QLP)

GENERAL PROCESS FLOW – TSOTQ first creates the GRMAP and GRDAT files. It then reads the following files to be used in translating response data and converting it into internal records: GTFLD, GFILN, GFILD, GTFDS, GTNAM, GFDES, and GADES. Function TOFNAM is called to process the QLP name recognition data. After the response data have been processed, TSOTQ writes a null Internal Record Map Header (GMPHD), waits for the DISPLAY process to terminate, and then exits. See Figure 49 for data flow.

### MAJOR FUNCTION DESCRIPTIONS

TOFNAM (Process Field Name Recognition Data) – TOFNAM searches for a field name, reading response strings as needed. If it recognizes end-of-data, it stores any occurrence nodes which have been saved in a stack into GRMAP, then writes out the record map header (GMPHD), record map (GRMAP), and final record data block (GRDAT) and returns. If TOFNAM has found a field name, it finds a match in the transformation field data (GTFDS) and saves the field's index and size. TOFNAM determines whether this field starts a new record and, if so, stores occurrence nodes in GRMAP and writes out the files as in the preceding. If the field has no data value, TOFNAM proceeds to the next field. If the field is not in the basic data set, TOFNAM saves the aggregate indexes of the field's parent aggregate and its superordinate aggregates, if any. If this is the first field encountered for its aggregate, an occurrence node is started in a stack and a dataset node is stored in GRMAP. If this field starts a new occurrence for its aggregate, the existing occurrence node in the stack is updated and a dataset node is stored in GRMAP. Pointers to the dataset nodes are stored in the occurrence node in the stack with the occurrence count. For all fields which have a data value, TOFNAM calculates an index into GRDAT. If the current GRDAT block is full, it is written out and a new block is started. TOFNAM reads the field's value(s) and stores the data in GRDAT. A pointer and occurrence count are stored in GRMAP for each multivalued and variable-length field. After processing this field, TOFNAM goes on to the next field. After the first record has been built, the DISPLAY process is activated via a fork/execute sequence.

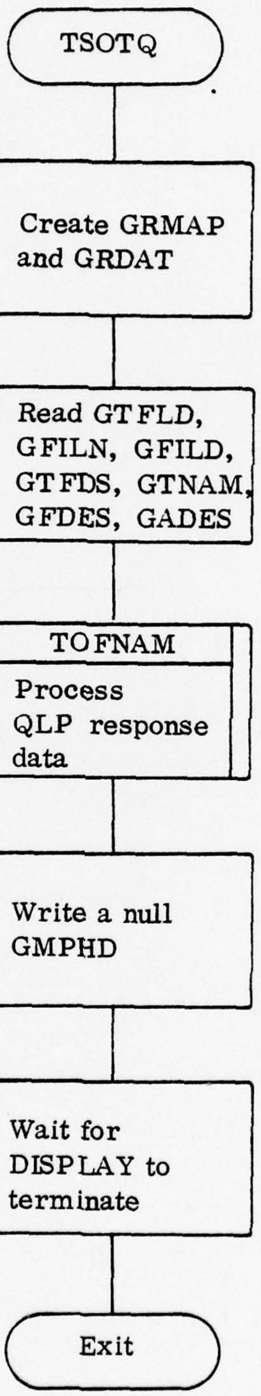

TSOTQ
Create GRMAP and GRDAT
Read GTFLD, GFILN, GFILD, GTFDS, GTNAM, GFDES, GADES
TOFNAM
Process QLP response data
Write a null GMPHD
Wait for DISPLAY to terminate
Exit

Figure 49. TSOTQ Process Data Flow (Sheet 1 of 3)

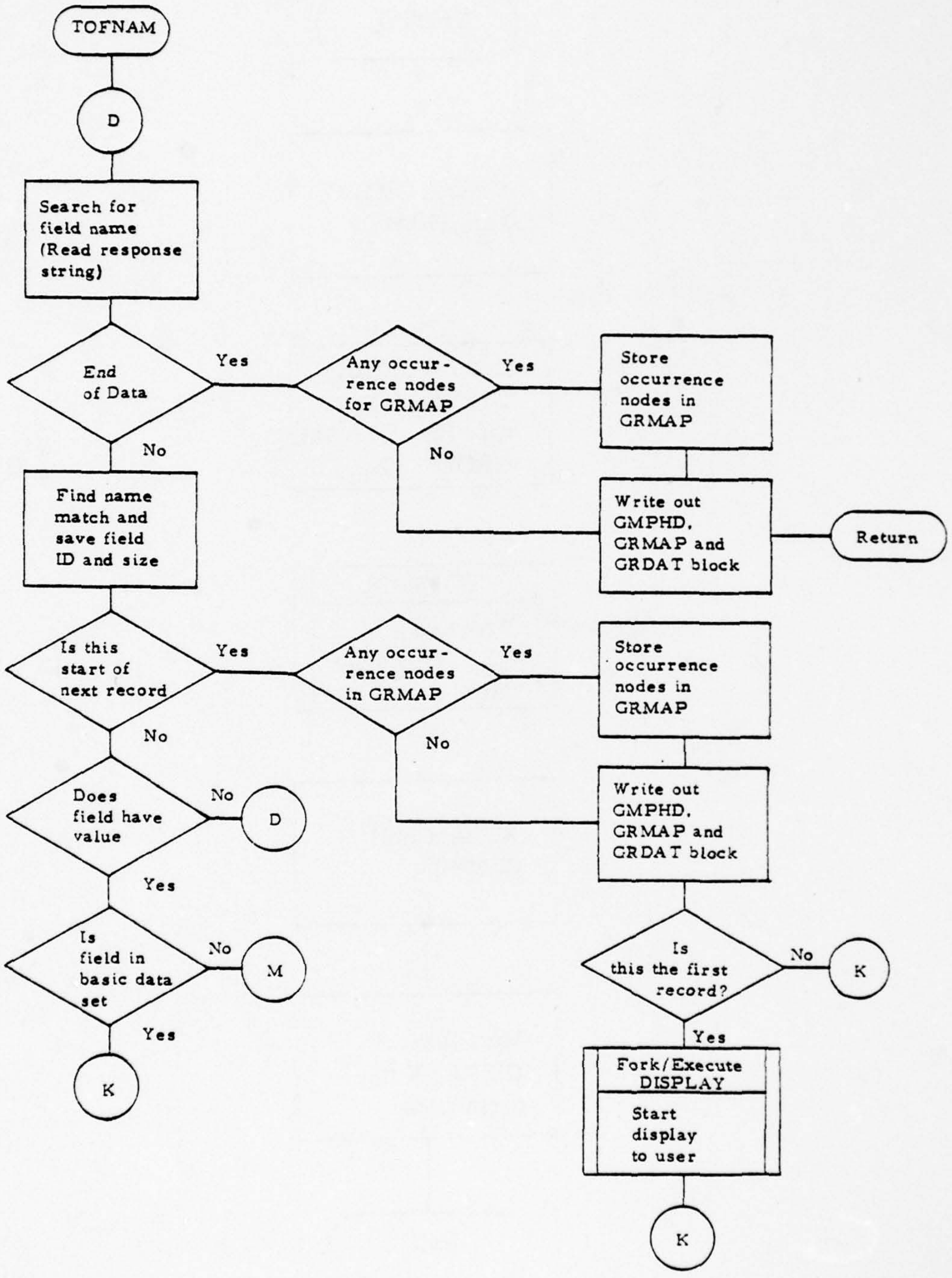

TOFNAM
D
Search for field name (Read response string)
End of Data
Yes
No
Any occurrence nodes for GRMAP
Yes
No
Store occurrence nodes in GRMAP
Write out GMPHD, GRMAP and GRDAT block
Return
Find name match and save field ID and size
Is this start of next record
Yes
No
Any occurrence nodes in GRMAP
Yes
No
Store occurrence nodes in GRMAP
Write out GMPHD, GRMAP and GRDAT block
Does field have value
No
D
Yes
Is field in basic data set
No
M
Yes
K
Is this the first record?
No
K
Yes
Fork/Execute DISPLAY
Start display to user
K

Figure 49. TSOTQ Process Data Flow (Sheet 2 of 3)

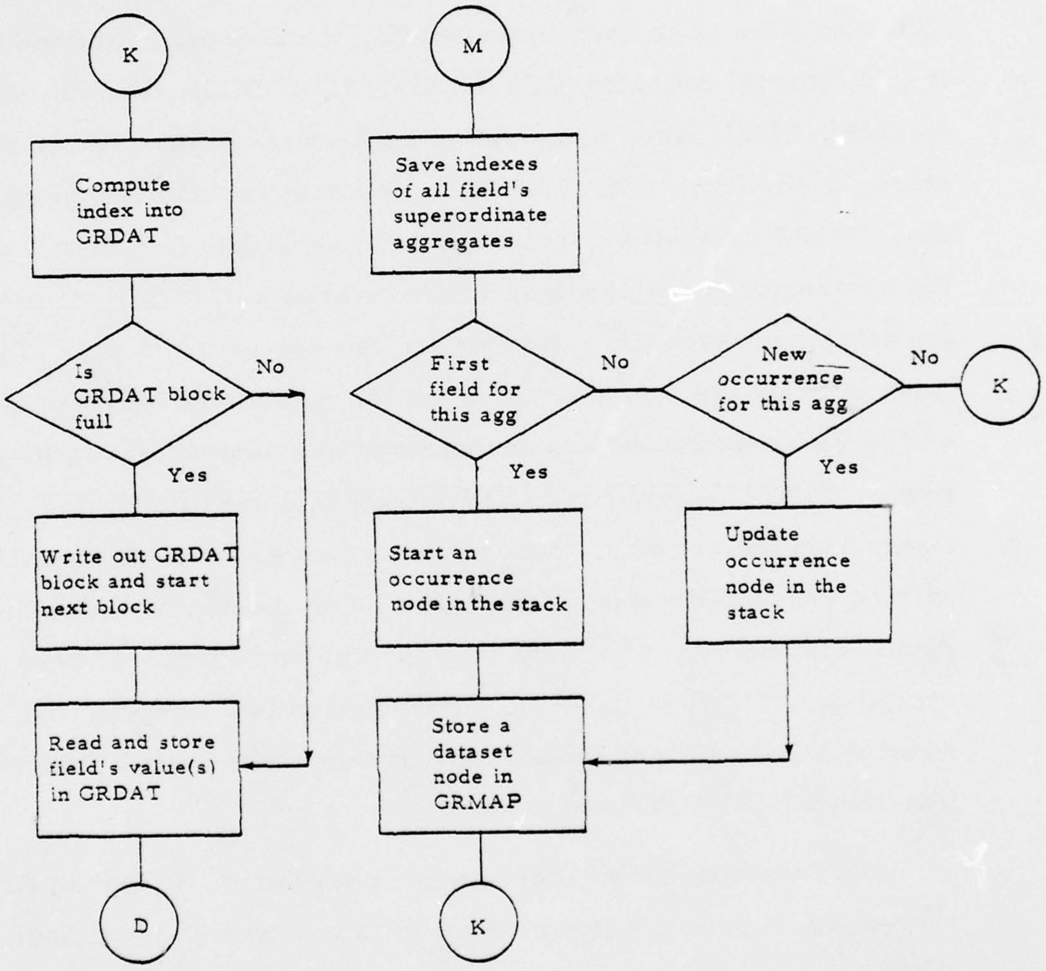

K
Compute index into GRDAT
Is GRDAT block full
No
Yes
Write out GRDAT block and start next block
Read and store field's value(s) in GRDAT
D
M
Save indexes of all field's superordinate aggregates
First field for this agg
No
Yes
New occurrence for this agg
No
K
Yes
Start an occurrence node in the stack
Update occurrence node in the stack
Store a dataset node in GRMAP
K

Figure 49. TSOTQ Process Data Flow (Sheet 3 of 3)

## TSOTR (TSOT for RYETIP)

GENERAL PROCESS FLOW – TSOTR picks up the GRMAP and GRDAT file descriptors passed by NETRESR. It then reads the following files to be used in translating the response data and converting it into internal records: GFILN, GFILD, GTFLD, GPFDS, GFDES, and GADES. TSOTR then picks up the whole record/partial data flag as stored in the first item of GPFDS. If whole records are being processed, then TSOTR computes the number of lines of data per record based on the number of characters per record stored in GTFLD. If partial data are being processed, then the number of lines is set to one. TSOTR reads lines of response data searching for the number of records line. If the end of the response data is encountered and partial data is being processed, TSOTR is finished and terminates as specified below. If the end is encountered and whole records are being processed, then TSOTR exits with an abnormal status. Otherwise, once the number of records line is found, the number of records is picked up and saved. If zero records is specified, TSOTR is finished. TSOTR outputs a line with the record count to the temporary output file. It then writes a null Internal Record Map Header (GMPHD) and exits.

After the non-zero record count is picked up, if partial data are being processed, a print statement number is incremented (originally zero). TSOTR positions the response data past two blank lines following the data header and then reads the next response line. If the line is END OF RUN and whole records are being processed, TSOTR is finished and terminates as specified above. If the line is END OF RUN and partial data is being processed, then TSOTR returns to the above loop searching for another number of records line. If the line is recognized as an error message, then TSOTR also returns to the above loop searching for another number of records line. Also, if partial data are being processed and there is not another record to process, TSOTR again returns to the same loop. Otherwise, TODATA is called to process the current record's data. If whole records are being processed, then TSOTR positions the response data past the blank line between records. TSOTR then continues with the smaller loop reading response strings and checking for END OF RUN. See Figure 50 for data flow.

## MAJOR FUNCTION DESCRIPTIONS

TODATA (Process a RYETIP Record) – If whole records are being processed or this is the first set of partial data, TODATA initializes the GRMAP and GRDAT buffers. Otherwise, TODATA reads in the GRMAP and GRDAT for the current record. If whole records are being processed, the remaining number of lines of response data are read into the buffer (TOBUFF). TODATA then processes each field in the referenced file as follows. The field index is picked up. If partial data is being processed, and the print statement number for the field as stored in GPFDS does not match the current print statement number, TODATA continues with the next field. Otherwise, the field's start character position is picked up from GPFDS. The field's data is stored in GRDAT, even if all blanks. If a field is multi-valued, only non-blank values are stored after the first value. After all fields have been processed, then TODATA writes out GRMAP and GRDAT and returns.

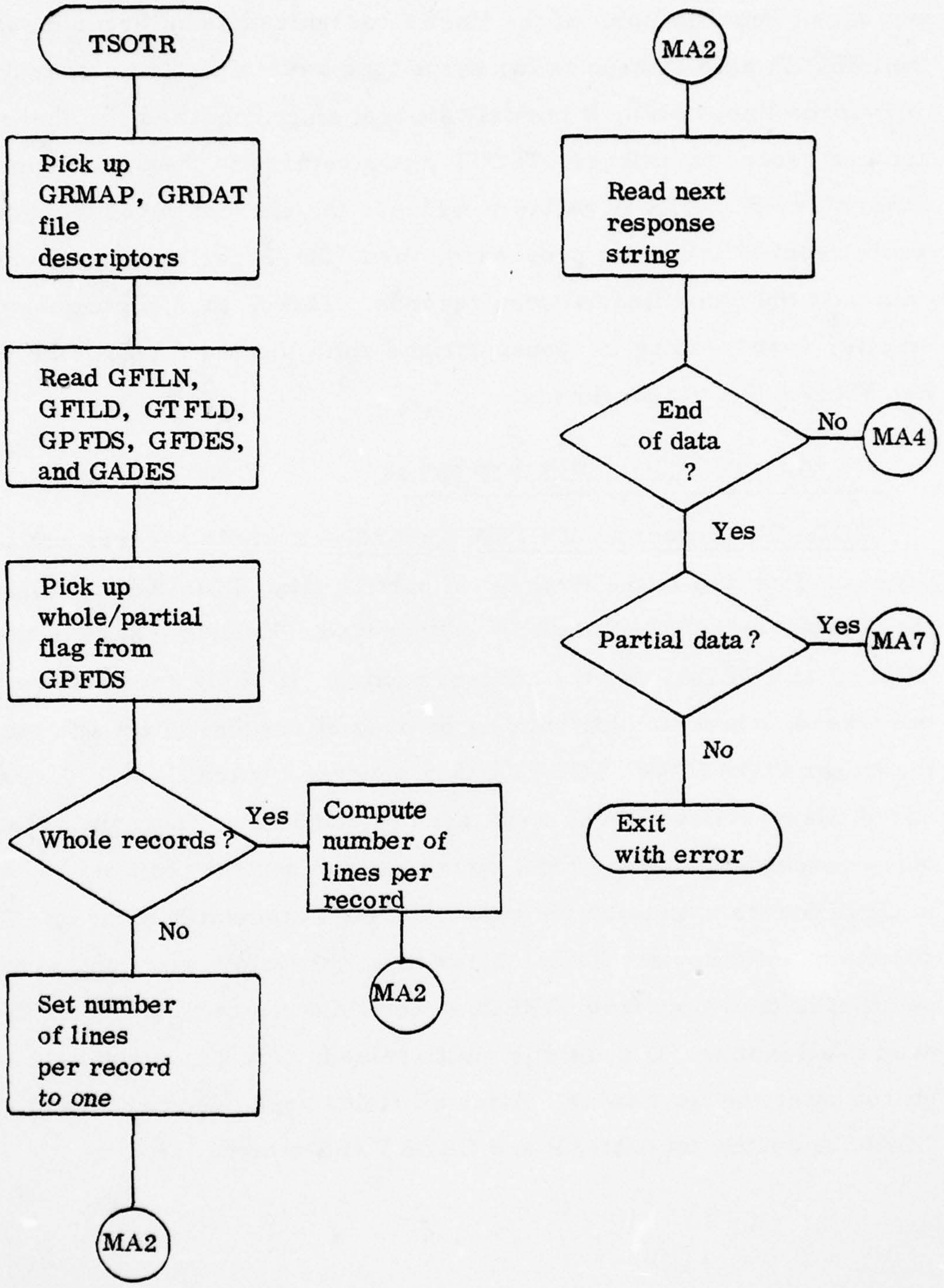

TSOTR
Pick up GRMAP, GRDAT file descriptors
Read GFILN, GFILD, GTFLD, GPFDS, GFDES, and GADES
Pick up whole/partial flag from GPFDS
Whole records ?
Yes
Compute number of lines per record
MA2
No
Set number of lines per record to one
MA2
MA2
Read next response string
End of data ?
No
MA4
Yes
Partial data ?
Yes
MA7
No
Exit with error

Figure 50. TSOTR Process Data Flow (Sheet 1 of 5).

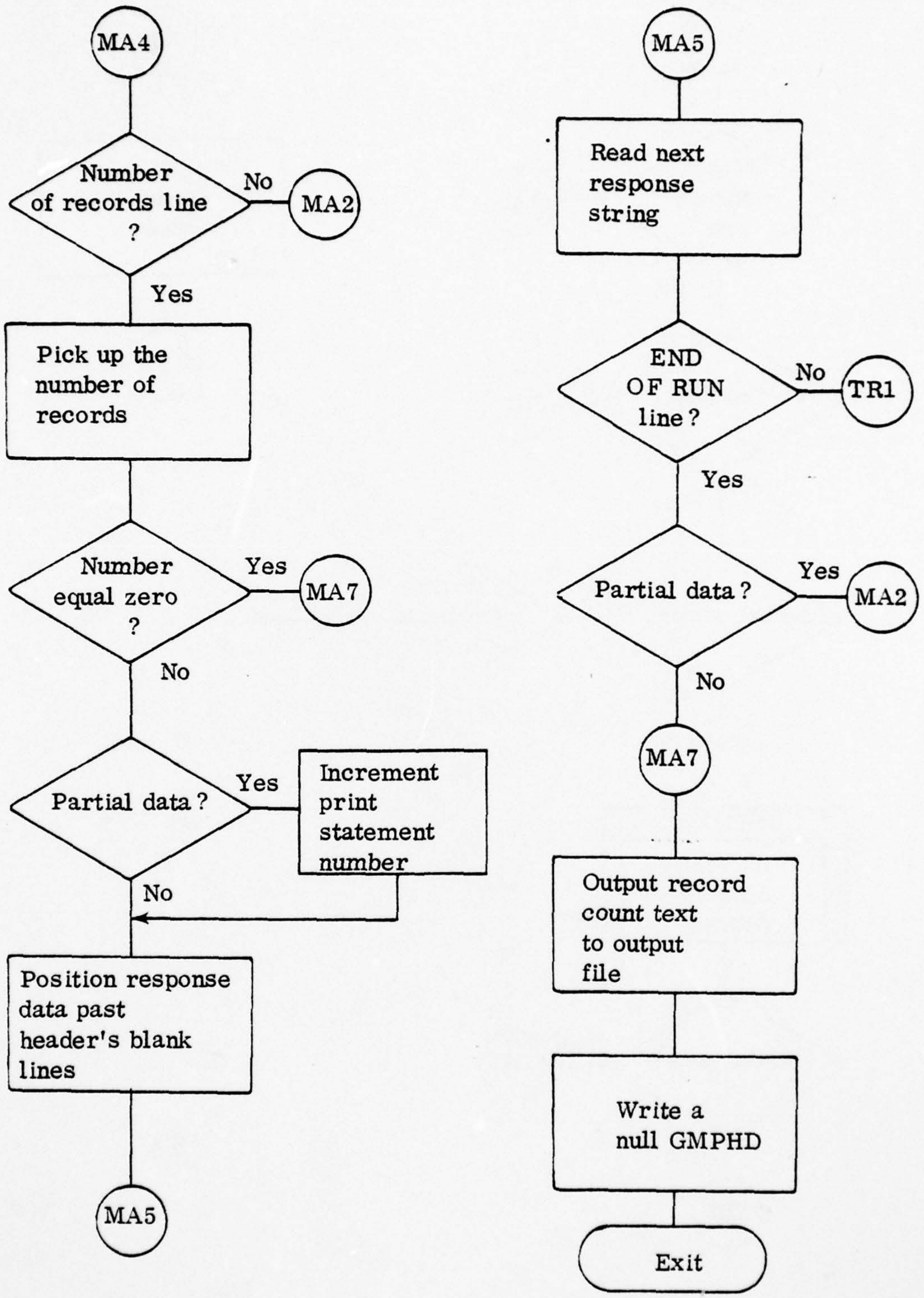

MA4
Number of records line ?
No
MA2
Yes
Pick up the number of records
Number equal zero ?
Yes
MA7
No
Partial data ?
Yes
Increment print statement number
No
Position response data past header's blank lines
MA5
MA5
Read next response string
END OF RUN line ?
No
TR1
Yes
Partial data ?
Yes
MA2
No
MA7
Output record count text to output file
Write a null GMPHD
Exit

Figure 50. TSOTR Process Data Flow (Sheet 2 of 5).

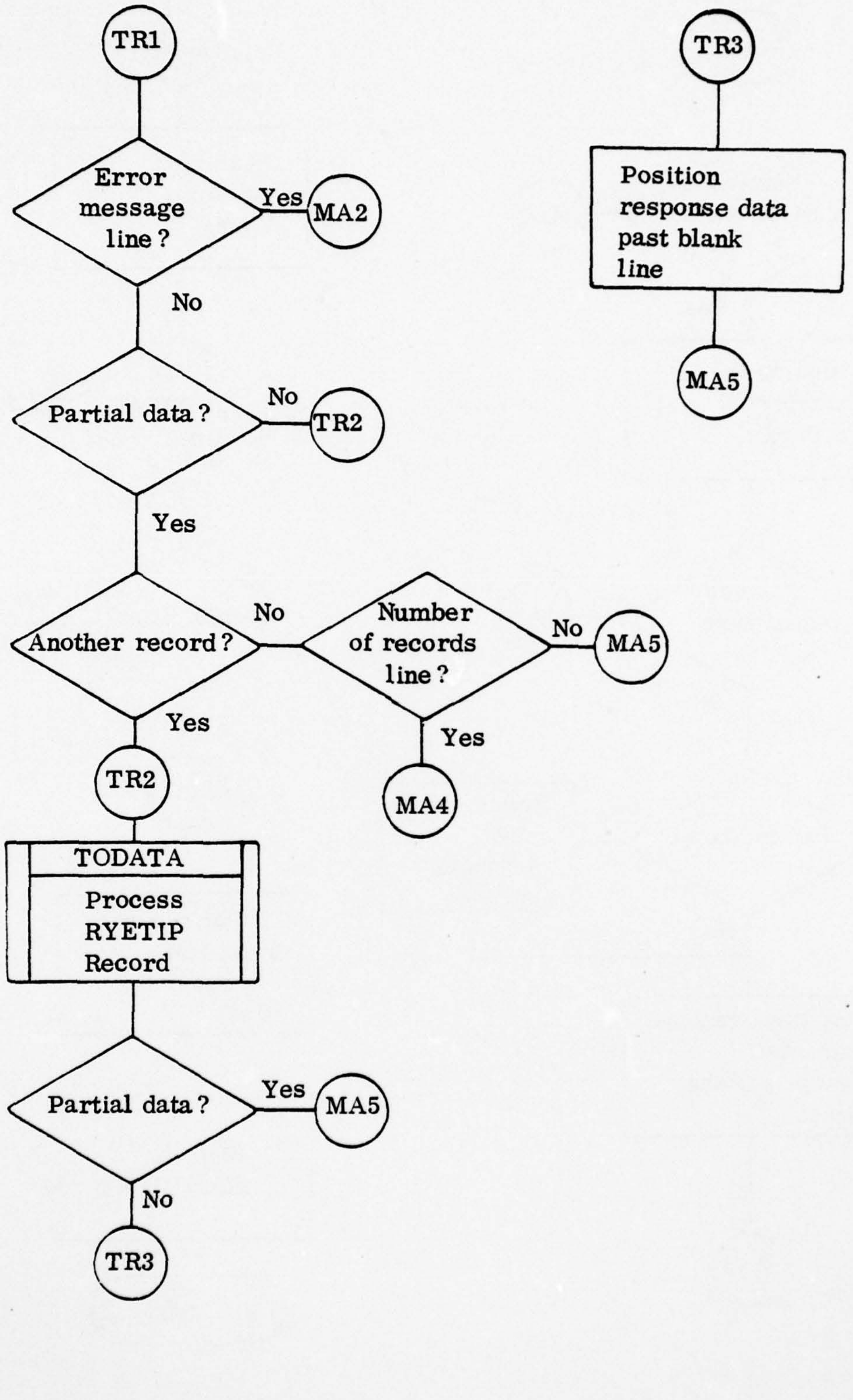


Figure 50. TSOTR Process Data Flow (Sheet 3 of 5).

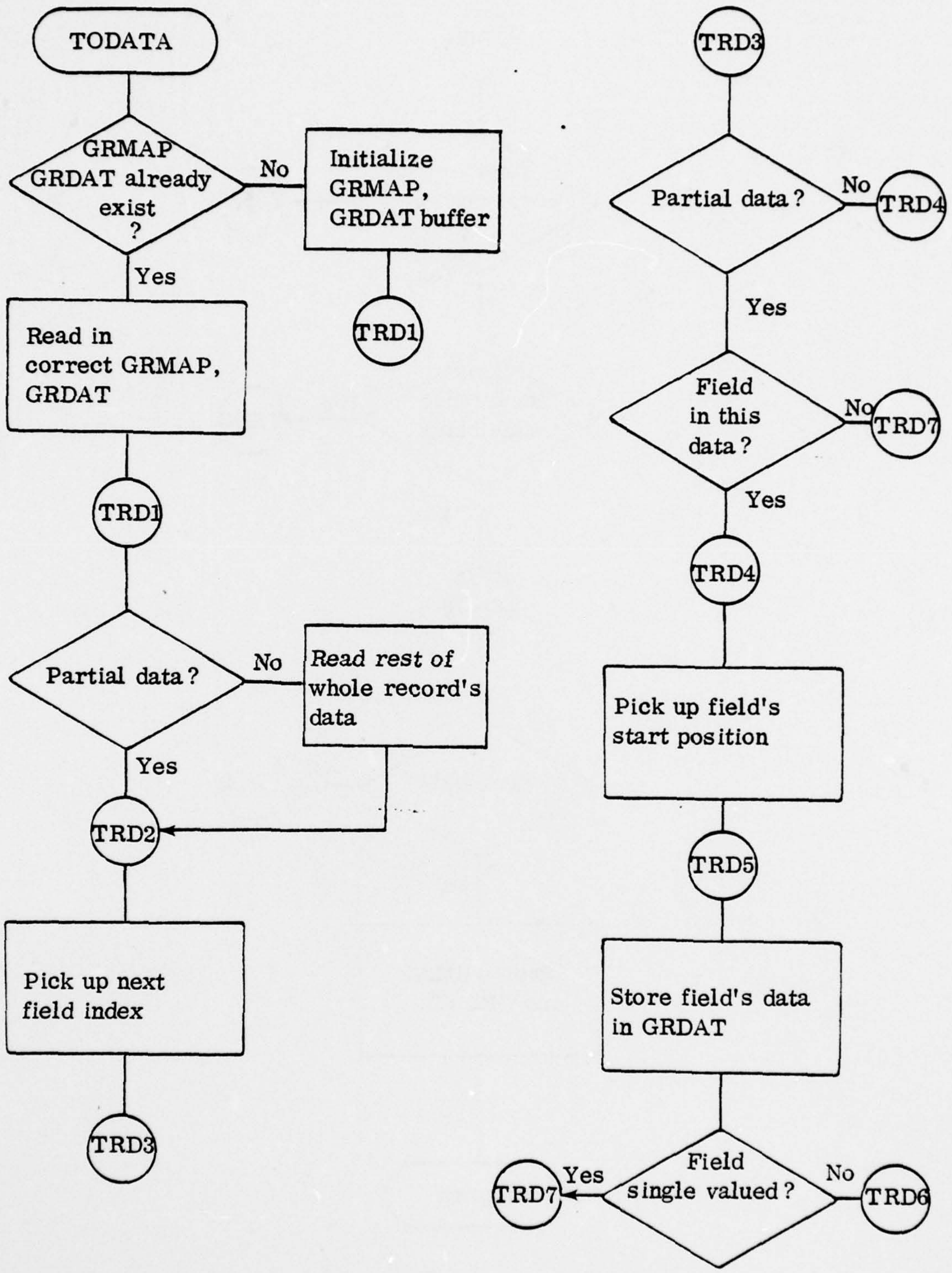

TODATA
GRMAP GRDAT already exist ?
No
Initialize GRMAP, GRDAT buffer
TRD1
Yes
Read in correct GRMAP, GRDAT
TRD1
Partial data ?
No
Read rest of whole record's data
Yes
TRD2
Pick up next field index
TRD3
TRD3
Partial data ?
No
TRD4
Yes
Field in this data ?
No
TRD7
Yes
TRD4
Pick up field's start position
TRD5
Store field's data in GRDAT
Field single valued ?
Yes
TRD7
No
TRD6

Figure 50. TSOTR Process Data Flow (Sheet 4 of 5).

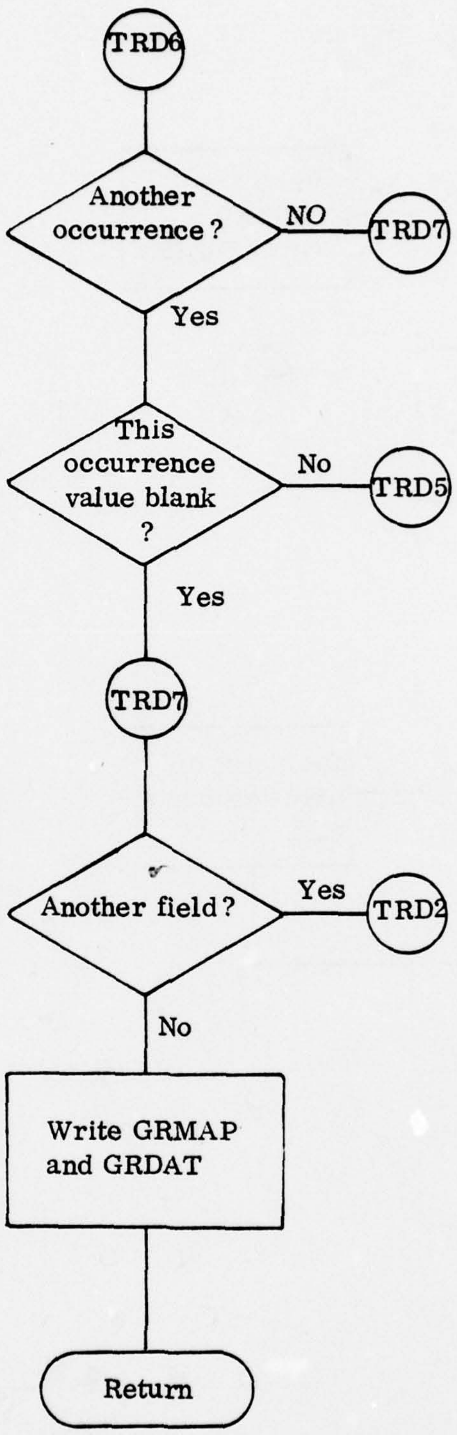


Figure 50. TSOTR Process Data Flow (Sheet 5 of 5).

## TSOTS (TSOT for SOLIS)

GENERAL PROCESS FLOW – TSOTS first creates the GRMAP and GRDAT files. It then reads the following files to be used in translating the response data and converting it into internal records: GFILN, GFILD, GTFDS, GTNAM, GFDES and GADES. TSOTS then determines whether text (message) data have been requested.

If text data are to be processed, TSOTS sends an SS command to SOLIS. TSOTS then reads the next screen of data. If both text and Preamble A0 are being processed, and this is not the first record, TSOTS sends a request for the specific message number wanted, i.e., M(number), and once again reads a screen of data. TSOTS searches for the Mode Selective Scan line. If this line cannot be found or if a SORRY screen is found, TSOTS assumes that all data has been processed. It then writes a null Internal Record Map Header (GMPHD), waits for the DISPLAY process to terminate and then exits. If the MODE line is found, then TSOTS searches for the line containing the message and page numbers. If this is the first record to be processed, TSOTS saves the message number line for later tests. Function TODATA is called to process this record's text data. If only text data are to be processed, and this is the first record, TSOTS performs a fork/execute sequence to activate DISPLAY. If only text data are being processed, TSOTS searches for the message number line on the current screen. If the message number line is not the same as the one saved from the first record, TSOTS processes the record's text data as specified above. If the message is the same as the first one processed, then the screens have wrapped around and TSOTS has processed all data. At this point, TSOTS terminates as specified above when the SORRY screen is found.

If Preamble A0 is to be processed as well as text, or if only Preamble A0 is to be processed, TSOTS sends an A0 command to SOLIS. TSOTS then reads the next screen of data. If both text and Preamble A0 are being processed, and this is not the first record, TSOTS sends a request for the specific message number wanted, i.e., M(number), and once again reads a screen of data. TSOTS searches for the message number line. If this is the first data of the first record to be processed, TSOTS saves the message number line for later tests. If this is not the first record, and the message is the same as the first one processed, the screens have wrapped around and TSOTS has processed all data. At this point, TSOTS terminates as specified above when the SORRY screen was found. Otherwise, TSOTS calls TOFNAM to process the name recognition data on the Preamble A0. If this was the first record, TSOTS performs a fork/execute sequence to activate DISPLAY. If both text and Preamble A0 are being processed, TSOTS returns to the beginning of the loop where an SS command is sent to SOLIS. If only Preamble A0 is being processed, then TSOTS continues in the smaller A0 loop and reads the next screen of data. See Figure 51 for data flow.

## MAJOR FUNCTION DESCRIPTIONS

TODATA (Process SOLIS Text) – TODATA is called with the first page of a message already read into the buffer (TOBUFF). TODATA saves the message number line for later tests. The field ID for the text field is determined by a search of GFDES. The index to TOBUFF is positioned past the header data and then the text data are stored in GRDAT. Special formatting characters are skipped over and the final character on each page of text, a caret '∧', is converted to a newline character.

When a block of GRDAT is filled up, it is written out and a new block is begun. After a page is processed, TODATA reads the next screen of data. TODATA searches for the message number line. If the message number line is the same as the first message number line saved or the message number is different from the one being processed by TODATA, GRDAT (and GRMAP if only text is being processed) is written out and TODATA returns. Otherwise, TODATA processes this page of data as specified above.

<u>TOFNAM (Process SOLIS Preamble A0)</u> – TOFNAM is called with a Preamble A0 screen already read into TOBUFF. TOFNAM reads strings of characters and compares them with the field names in GTNAM until a field name match is found. TOFNAM reads to the equals sign '=' and then determines if there is a value for the field. If there is, TOFNAM checks GRDAT to see if the field already has some data stored for it. The field's data from the SOLIS screen are then stored in GRDAT up to and including the first newline character encountered. If there was data already stored, this information is appended to it. When the end of the screen is encountered, the record's GRMAP and GRDAT files are written out and TOFNAM returns.

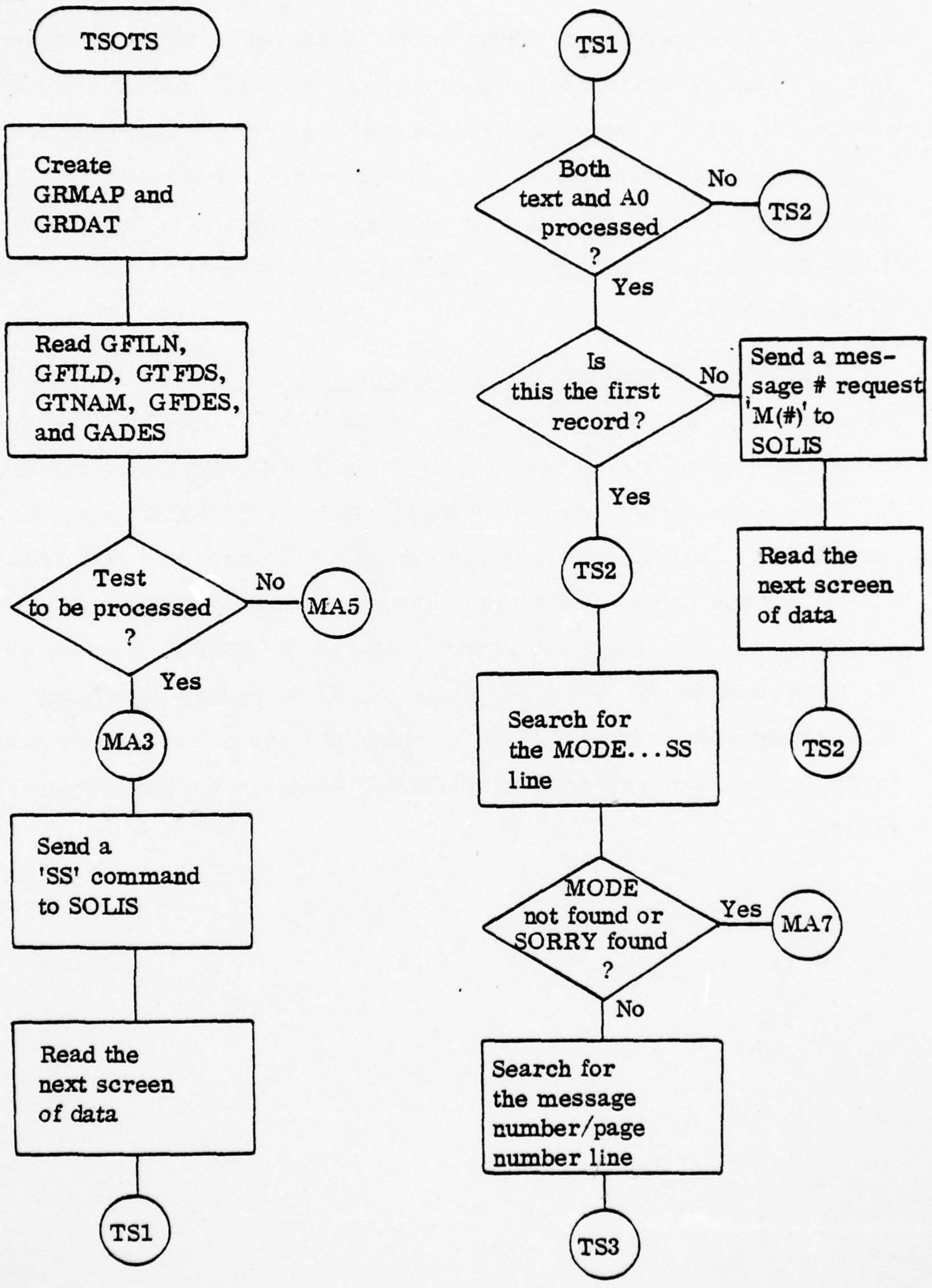
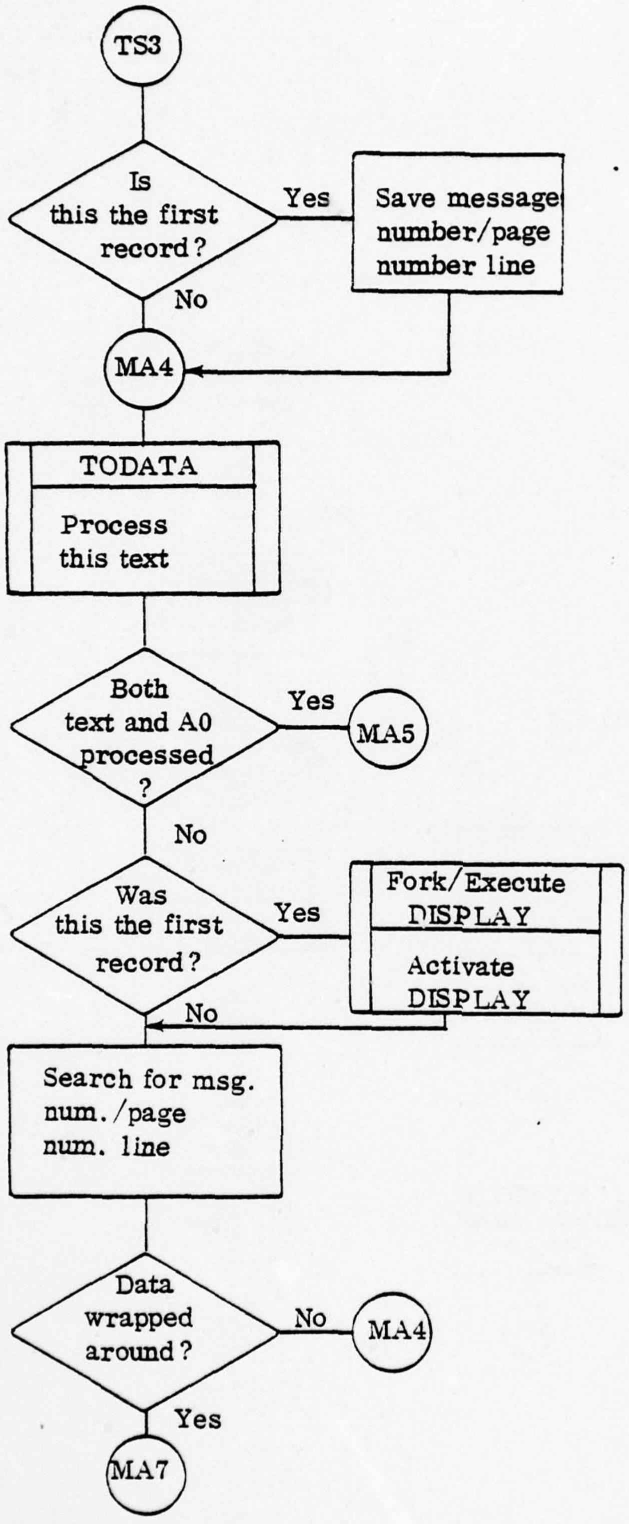

TSOTS
Create GRMAP and GRDAT
Read GFILN, GFILD, GTFDS, GTNAM, GFDES, and GADES
Test to be processed ?
No
MA5
Yes
MA3
Send a 'SS' command to SOLIS
Read the next screen of data
TS1
TS1
Both text and A0 processed ?
No
TS2
Yes
Is this the first record ?
No
Send a message # request 'M(#)' to SOLIS
Read the next screen of data
TS2
Yes
TS2
Search for the MODE...SS line
MODE not found or SORRY found ?
Yes
MA7
No
Search for the message number/page number line
TS3

Figure 51. TSOTS Process Data Flow (Sheet 1 of 5).


TS3
Is this the first record?
Yes
Save message number/page number line
No
MA4
TODATA
Process this text
Both text and A0 processed?
Yes
MA5
No
Was this the first record?
Yes
Fork/Execute DISPLAY
Activate DISPLAY
No
Search for msg. num./page num. line
Data wrapped around?
No
MA4
Yes
MA7

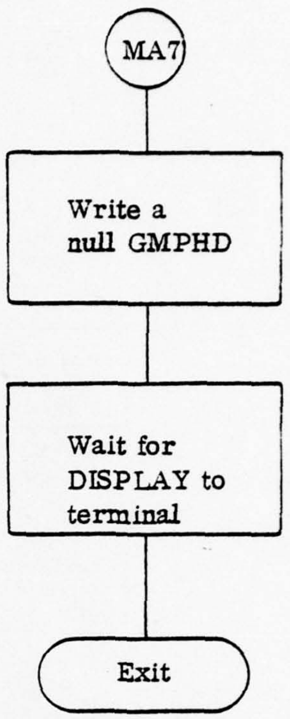


Figure 51. TSOTS Process Data Flow (Sheet 2 of 5).

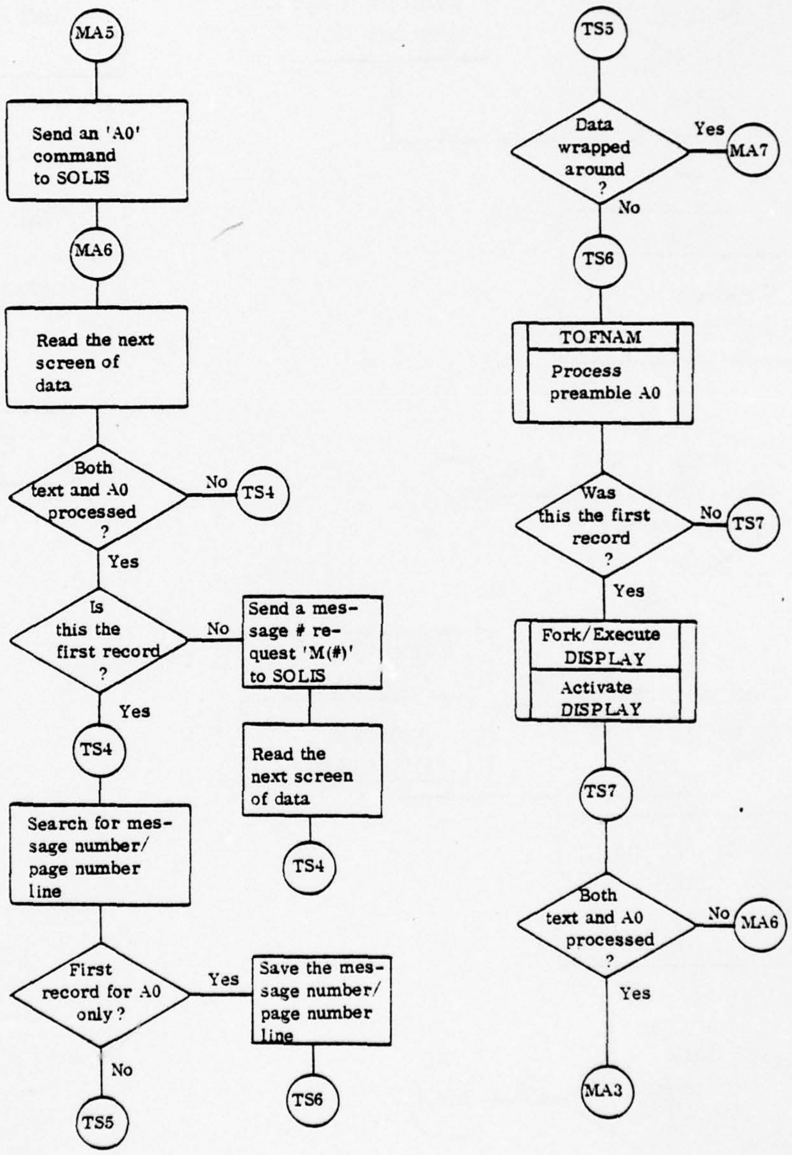
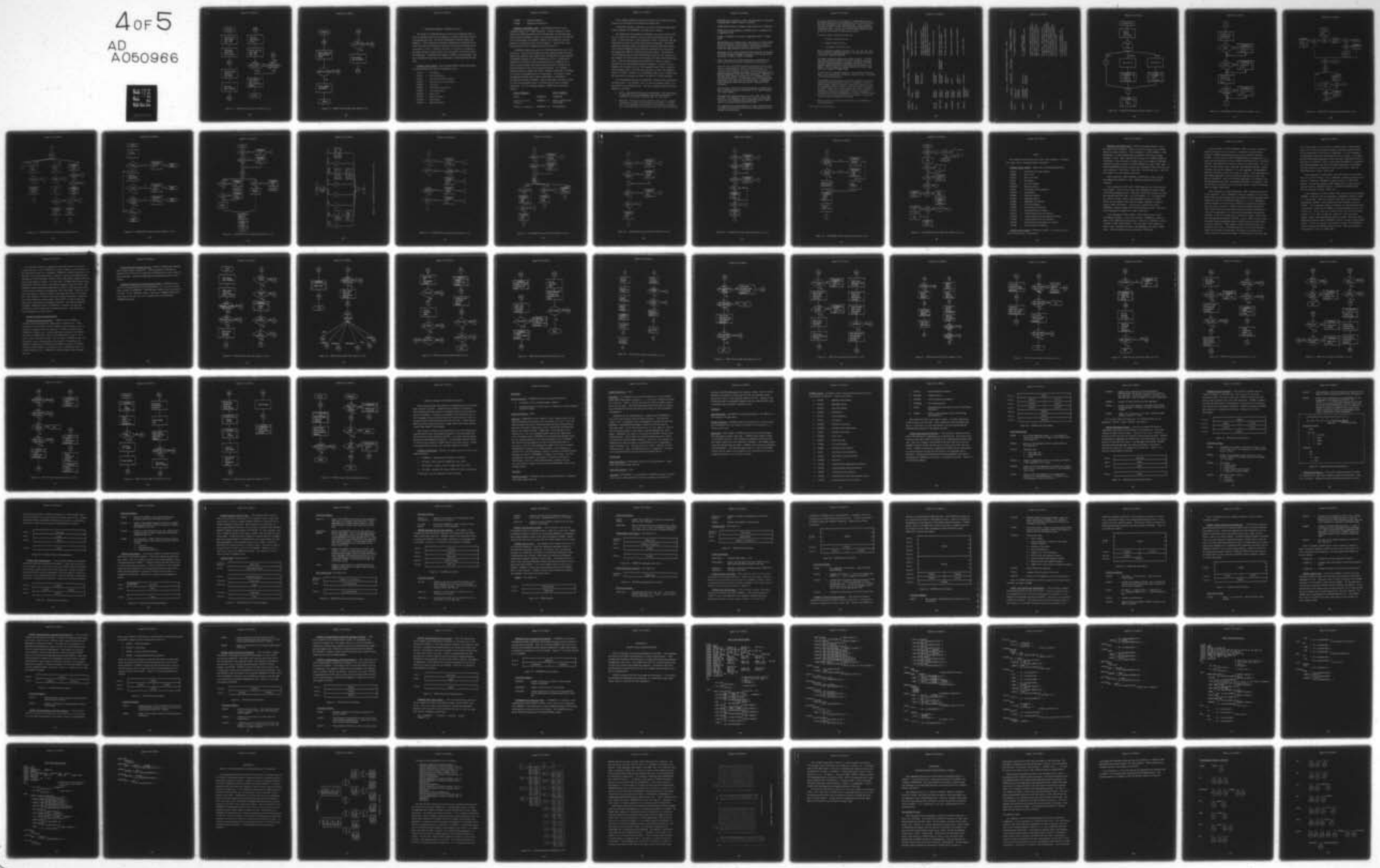

MA5
Send an 'A0' command to SOLIS
MA6
Read the next screen of data
Both text and A0 processed ?
No
TS4
Yes
Is this the first record ?
No
Send a message # request 'M(#)' to SOLIS
Read the next screen of data
TS4
Yes
TS4
Search for message number/ page number line
First record for A0 only ?
Yes
Save the message number/ page number line
TS6
No
TS5
TS5
Data wrapped around ?
Yes
MA7
No
TS6
TOFNAM
Process preamble A0
Was this the first record ?
No
TS7
Yes
Fork/Execute DISPLAY
Activate DISPLAY
TS7
Both text and A0 processed ?
No
MA6
Yes
MA3

Figure 51. TSOTS Process Data Flow (Sheet 3 of 5).

AD-A050 966    LOGICON INC SAN DIEGO CALIF    F/G 5/2
ADAPT I FINAL FUNCTIONAL AND SYSTEM DESIGN SPECIFICATION.(U)
JAN 78  L R ERICKSON, M E SOLEGLAD    N00014-76-C-0899
UNCLASSIFIED    76-C-0899-2-F    NL

4 OF 5
AD
A050966



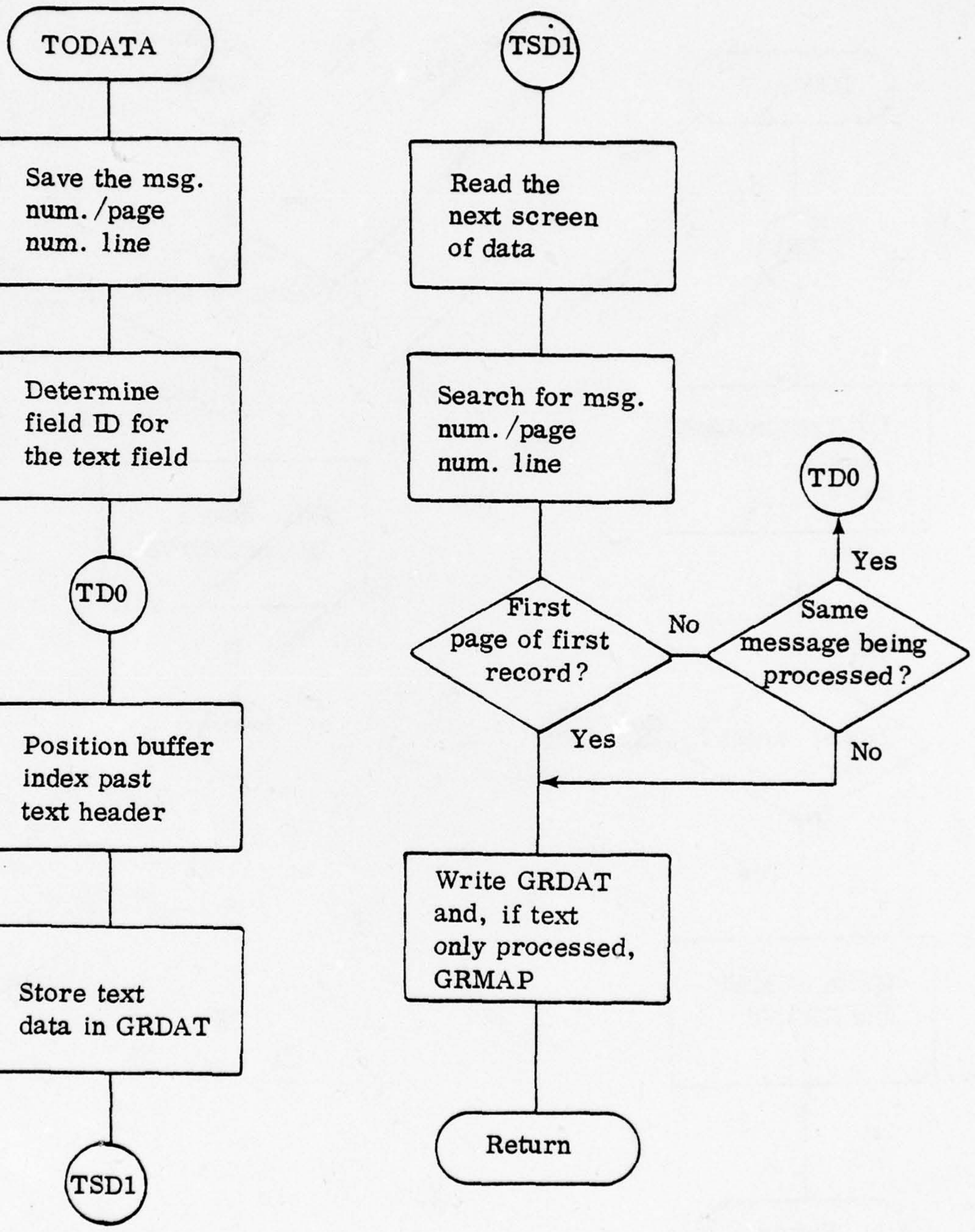

TODATA
Save the msg. num./page num. line
Determine field ID for the text field
TD0
Position buffer index past text header
Store text data in GRDAT
TSD1
TSD1
Read the next screen of data
Search for msg. num./page num. line
First page of first record?
No
Yes
Same message being processed?
Yes
No
TD0
Write GRDAT and, if text only processed, GRMAP
Return

Figure 51. TSOTS Process Data Flow (Sheet 4 of 5).

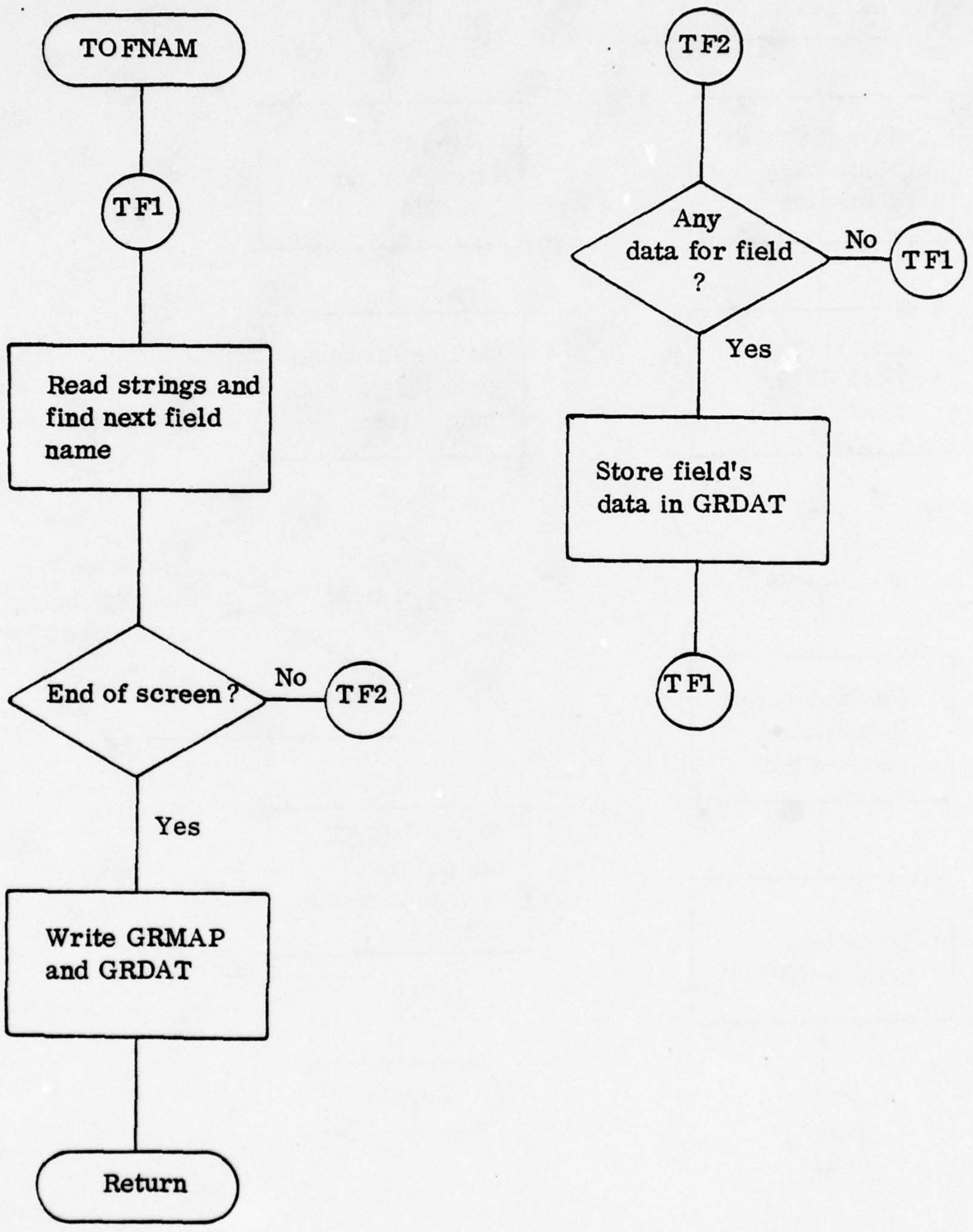


Figure 51. TSOTS Process Data Flow (Sheet 5 of 5).

## VALIDATE (Semantic Validation Process)

The ADAPT I VALIDATE process checks three statement types of UDL for semantic accuracy and legality according to information stored in the global data files. The three statements are SAVE, FIND, and DISPLAY. Each statement successfully validated by VALIDATE can be further processed with minimal concern for nonsensical statements and meaningless results. Semantic mistakes are located and diagnostic messages printed in order that the user can quickly correct and reenter the statement. VALIDATE also modifies GTREE, when applicable, with more meaningful token types and with indexes to appropriate global data files.

GLOBAL DATA USAGE – The following ADAPT I global data tables are utilized extensively by the VALIDATE process:

| | | |
|---|---|---|
| GFILN | – | file names. |
| GFILD | – | file descriptions. |
| GLIST | – | list names and descriptions. |
| GLABL | – | label names and descriptions. |
| GOPEN | – | open files. |
| GTDES | – | transaction descriptions. |
| GUSER | – | user descriptions. |
| GTREE | – | parse tree. |
| GTDAT | – | parse tree data. |
| GFNAM | – | field names. |
| GANAM | – | aggregate names. |

GFDES – field descriptions.

GADES – aggregate descriptions.

GENERAL PROCESS FLOW – VALIDATE performs semantic validation upon a user's input statement. Control is passed to VALIDATE when EXEC determines that a syntactically correct input statement requires validation. Seven of the arguments passed to VALIDATE are file descriptors for GTREE, GTDAT, user output file, and four pipes (two read and two write file descriptors). The two other arguments passed are the validation mode and the user ID.

VALIDATE reads in the parse tree (GTREE) and picks up the first token type. This token type defines the UDL statement type and subsequently the semantic rules against which VALIDATE tests the statement. VALIDATE continues to work through the parse tree, validating the statement in accordance with a prespecified set of validation rules. After a valid terminal token type has been processed, the parse tree may be modified to more closely define the token. For example, a NAME located in a display list within a DISPLAY statement is validated to be either an aggregate name or a field name. If the name is legal, the appropriate token type of either AGGNAME or FLDNAME is exchanged for NAME in GTREE. Whenever possible, the name's index for its associated global table is inserted after the newly assigned token type. For the foregoing example, GTREE may be altered as follows:

| Before Validation | | After Validation |
|---|---|---|
| NAME | GTREE [n] | AGGNAME |
| Pointer to the first character | GTREE [n + 1] | Index to both GANAM and GADES |
| Number of characters | GTREE [n + 2] | Not applicable |

These indexes expedite further processing of the statement by other processes by allowing direct indexing into global files.

VALIDATE attempts to find all errors and to print meaningful diagnostic messages via GFERROR, the global error routine.

On completion of the statement validation, the process returns with an error value to the EXEC if an error had been detected in the statement. On the other hand, if the statement is valid, VALIDATE calls the appropriate process to continue processing the statement. The FIND statement selection criteria are logically normalized and minimized by BEN, while the DISPLAY (tag) and SAVE statements are processed by the appropriate TSIG for the given target host system. The DISPLAY (list) statement causes access authorization checks for the clearance of the response for both the user and terminal to be done by the TAS process AAP. If the user and terminal are authorized, a check is made on whether the user had already viewed the response. If the user had not displayed the response yet, the TAS Logging Process (TLP) is called to log the display event. Once the event has been recorded, the DISPLAY process is called. However, if the validation mode is set to validate only, the only other processing to be done is by BEN for a FIND statement.

The remaining discussion of VALIDATE is broken down by individual statement types. The semantic rules, as well as program details, are discussed for each statement type. The three statement types are SAVE, DISPLAY, and FIND.

a. SAVE – The token SAVE points to two NAMEs. The first NAME is validated to be an existing label (tag); the second NAME is validated to be neither an existing label nor a list name.

b. DISPLAY – The first leg of the DISPLAY token points to NAME, confirmed to be either an existing tag or list name. (The DISPLAY (tag) statement requires that the specified file is opened.) The second leg (if there is one) points to a token type of DISPLIST.

DISPLIST has any number of legs, each pointing to a token type of CHARCONS, TREE, NAME, or SNAME.

CHARCONS (character constant string) requires no validation.

TREE has one leg pointing to a NAME which is validated to be an aggregate name.

NAME is validated to be either an aggregate name or a field name.

SNAME points to a NAME which is checked to be a subscripted field whose number of subscripts is the number of levels of arrays in which the field is located. Each subscript must be within the range defined for the associated array.

Additional checks require that the display attribute of all fields be visible and that the display list have at least one leg whose token type is TREE, NAME, or SNAME.

c. FIND – The label on the FIND statement is validated to be a name different from already existing tag and list names.

The first leg of the FIND statement must specify either a file name or a label name of a previous FIND statement; the token is either IN or SOURCE. The name to which IN points is required to be an existing file name, and the statement is completely independent of previous FIND statements. SOURCE, on the other hand, points to a name which is validated to be an existing tag (GLABL). The statement itself is dependent upon a previous query labelled by the specified tag. A dependent FIND statement cannot reference another dependent FIND statement.

The selection-criteria of the FIND statement is nothing more than a specialized boolean expression for the explicit purpose of record selection.

The legal UDL boolean operators are OR, AND, XOR, NOR, and NOT. No validation is done at this token level although, at the end of the validation of the statement, boolean expressions are normalized and minimized by BEN.

The argument of a boolean operator is either a selection term or another selection-criteria enclosed in parentheses with an optional scope-qualifier.

A scope-qualifier must be the name of a repeating group, and the fields referenced in the qualified selection-criteria must be directly subordinate to the repeating group. Scope-qualifiers can also be nested to force the desired lineage path during the selection process. Again, those nested scope-qualifiers must be directly subordinate repeating group names.

The three types of relational terms are:

1. Standard relational terms.

2. Range relational terms.

3. Geographic relational terms.

The five standard relational operators LT, GT, LE, GE, and EQ are numeric relational operators. EQ can also be used as a character relational operator.

The range relational operators are WRG and ORG. The WRG operator requires the field to be within or equal to the limits set by the two given numeric constants. Similarly, the ORG operator requires the field to be outside the two numeric constants. The first numeric constant specifies the lower range limit; the second, the upper limit.

The last type of relational operators, the geographic operators, apply to the relationships of fields to circles, routes, or polygons on the surface of the earth.

The INSIDE operator requires the field's geographic value to be inside the area specified by a circle or a polygon. The OUTSIDE operator requires the field's value to be outside the area of a specified polygon. Finally, the ALONG operator requires the field's value to be on the perimeter of a route. CIRCLE is defined by a radius and a geographic constant, and POLY is designated by a series of geographic constants. ROUTE is established by both the series of geographic constants on which the route lies and the bandwidth of the route.

Refer to table 1 for more specific details on the validation of FIND statements.

See Figure 52 for data flow.

TABLE 1. SELECTION-CRITERIA VALIDATION.

| Keyword | First Leg | Second Leg | Third Leg | Explanation of Validation |
|---|---|---|---|---|
| SCOPE | NAME | — | — | Aggregate name of a repeating group. |
| EQ | NAME or SNAME | NUMCONS or INTEGER or | — | Field's data type is numeric. |
| | | CHARCONS | — | Field's data type is character. Unless field type is single-valued variable, the number of characters in CHARCONS is less than or equal to the field's size. |
| LT, LE, GT, GE | NAME or SNAME | NUMCONS or INTEGER | — | Field's data type is numeric. |
| WRG, ORG | NAME or SNAME | NUMCONS or INTEGER | NUMCONS or INTEGER | Field's data type is numeric, first number $\leq$ second number. |
| INSIDE | NAME or SNAME | CIRCLE or POLY | — | Field's data type is geographic. |
| OUTSIDE | NAME or SNAME | POLY | — | Field's data type is geographic. |
| ALONG | NAME or SNAME | ROUTE or POLY | — | Field's data type is geographic. |
| CIRCLE | NUMCONS or INTEGER (radius) | GEOCONS | — | $0.1 \leq$ radius $\leq 2000$. |

TABLE 1. SELECTION-CRITERIA VALIDATION. (Cont)

| Keyword | First Leg | Second Leg | Third Leg | Explanation of Validation |
|---|---|---|---|---|
| POLY | Pointer to a series of GEOCONS | – | – | $3 \leq$ number of GEOCONS $\leq 506$. |
| ROUTE | NUMCONS or INTEGER (bandwidth) | Pointer to a series of GEOCONS | – | $0.1 \leq$ bandwidth $\leq 2000$.<br>$3 \leq$ number of GEOCONS $\leq 506$. |
| NAME | – | – | – | Field name whose parent is not an array. Keyed field if tag is independent. Keyed or dependent if tag is dependent. If there is a scope-qualifier, it is the parent of the field. |
| SNAME | – | – | – | Field name. The number of subscripts is the correct levels of subscript for the field. Subscript number is within the occurrence size (GOCCS). Keyed field if tag is independent. Keyed or dependent if tag is dependent. If there is a scope-qualifier, it is the parent of the field. |
| GEOCONS | – | – | – | $0 \leq$ degrees for latitude $\leq 90$.<br>$0 \leq$ degrees for longitude $\leq 180$.<br>$0 \leq$ minutes $\leq 59$.<br>$0 \leq$ seconds $\leq 59$. |

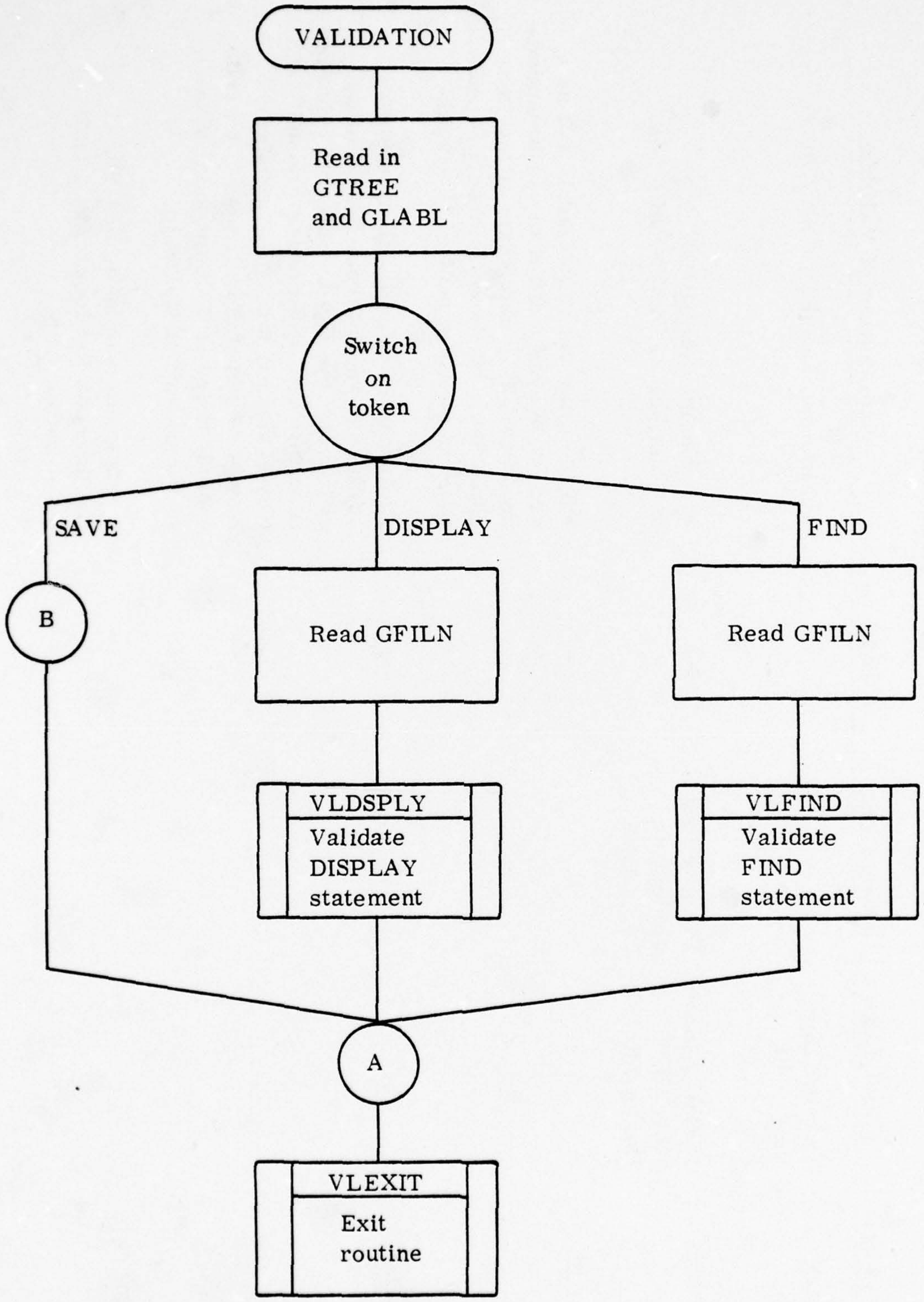


Figure 52. VALIDATE Process Data Flow (Sheet 1 of 13).

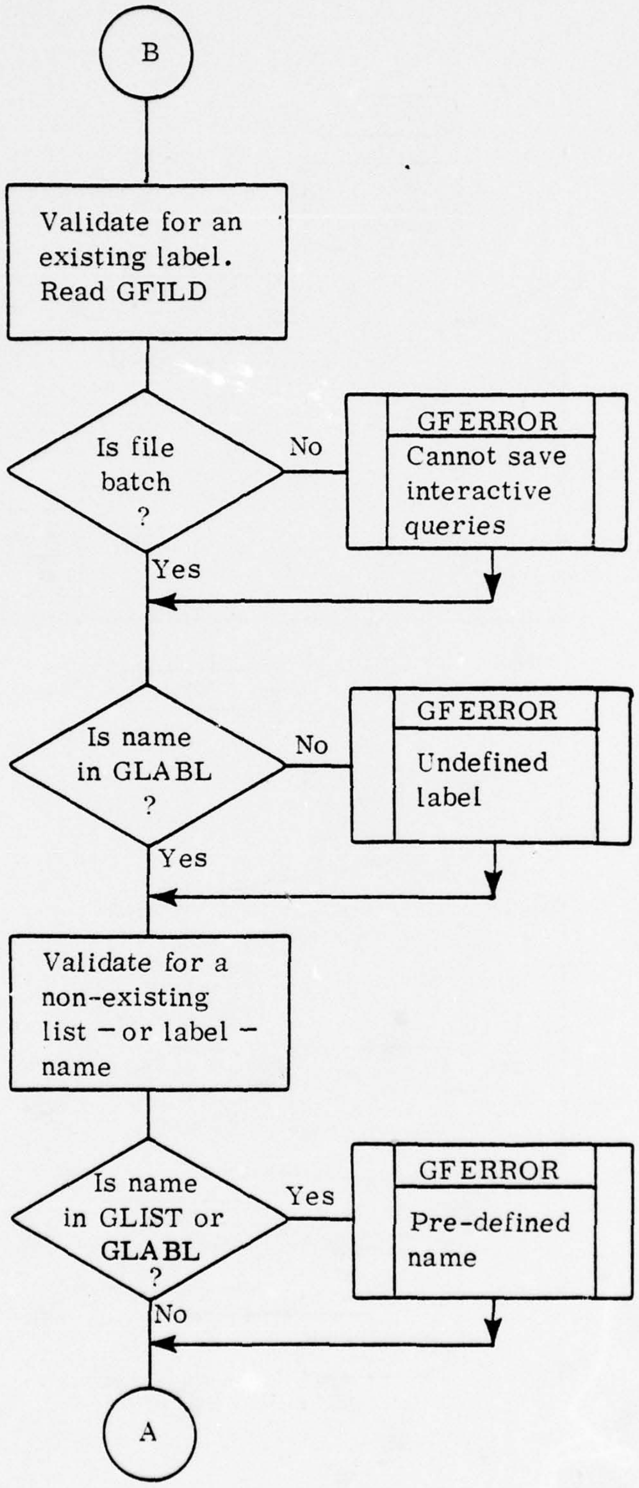

B
Validate for an existing label. Read GFILD
Is file batch ?
No
GFERROR
Cannot save interactive queries
Yes
Is name in GLABL ?
No
GFERROR
Undefined label
Yes
Validate for a non-existing list — or label — name
Is name in GLIST or GLABL ?
Yes
GFERROR
Pre-defined name
No
A

Figure 52. VALIDATE Process Data Flow (Sheet 2 of 13).

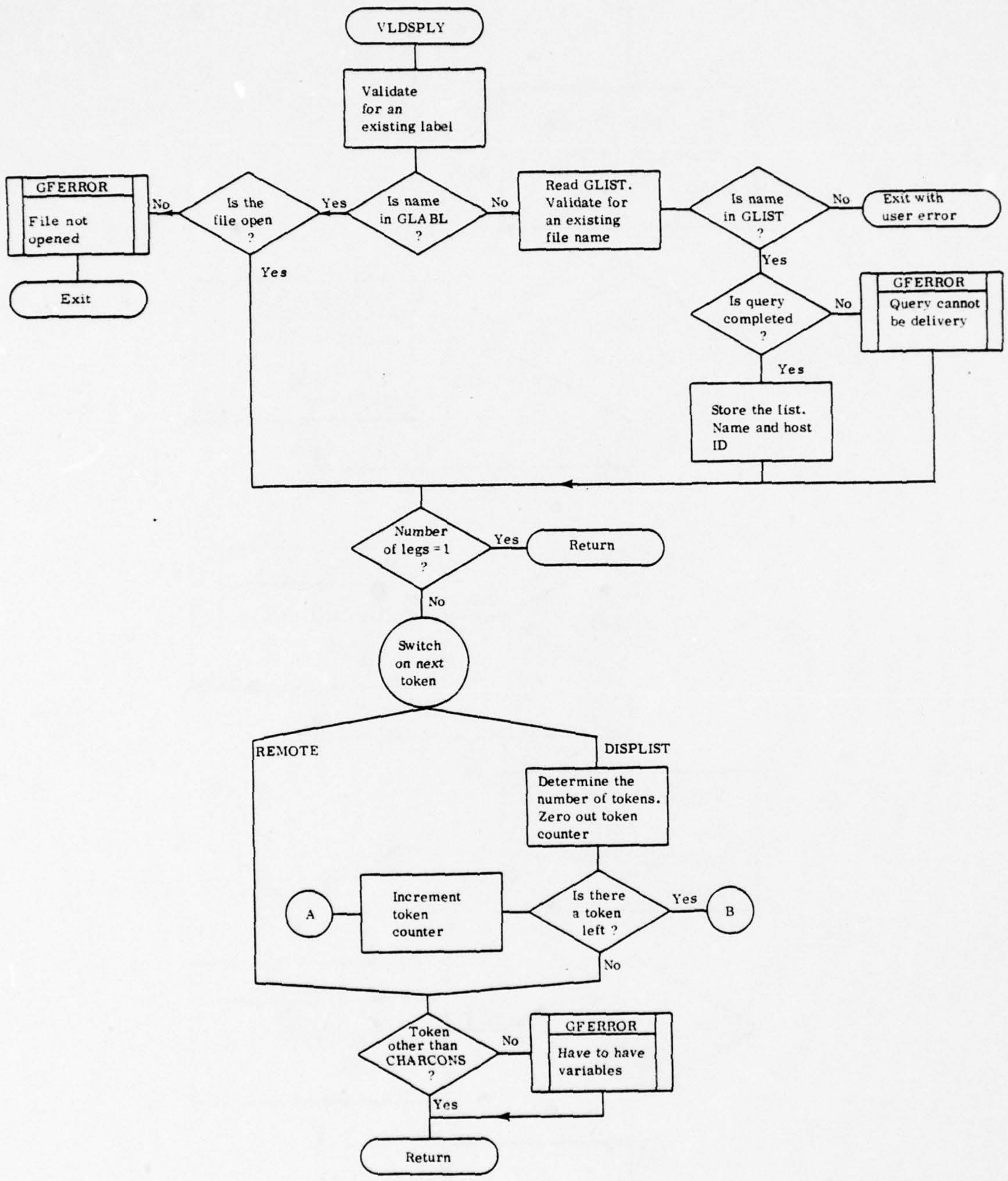


Figure 52. VALIDATE Process Data Flow (Sheet 3 of 13).

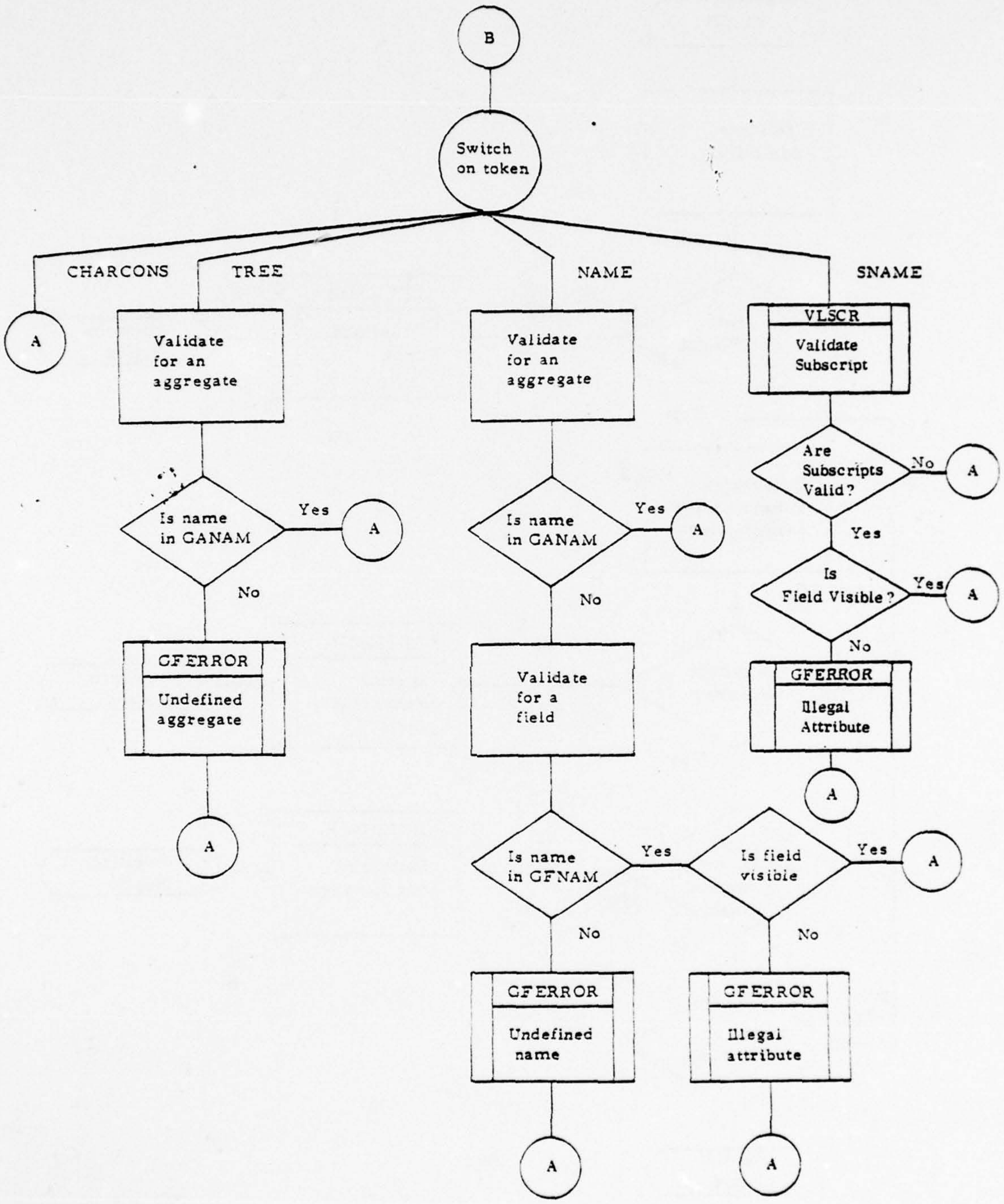


Figure 52. VALIDATE Process Data Flow (Sheet 4 of 13).

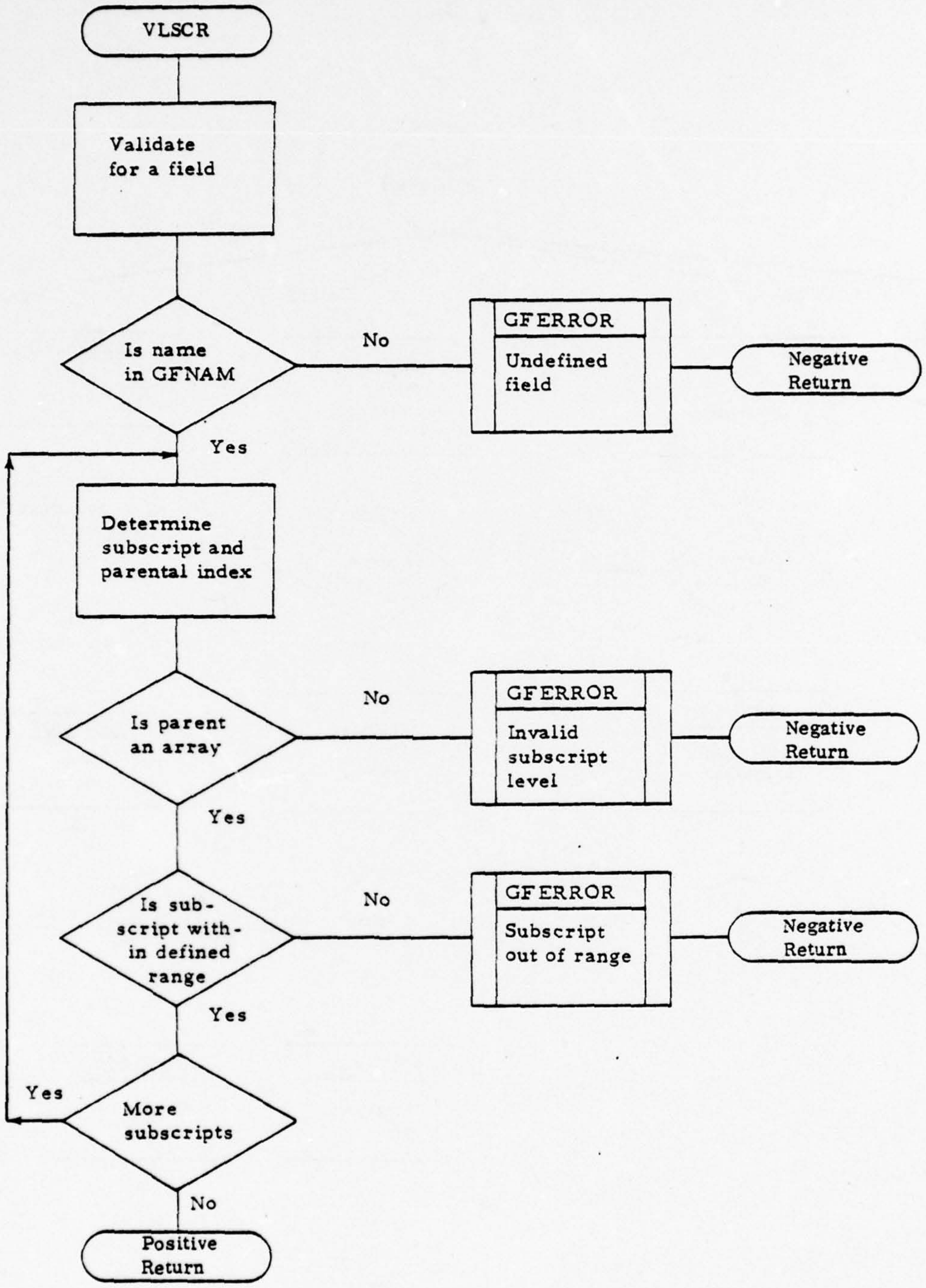


Figure 52. VALIDATE Process Data Flow (Sheet 5 of 13).

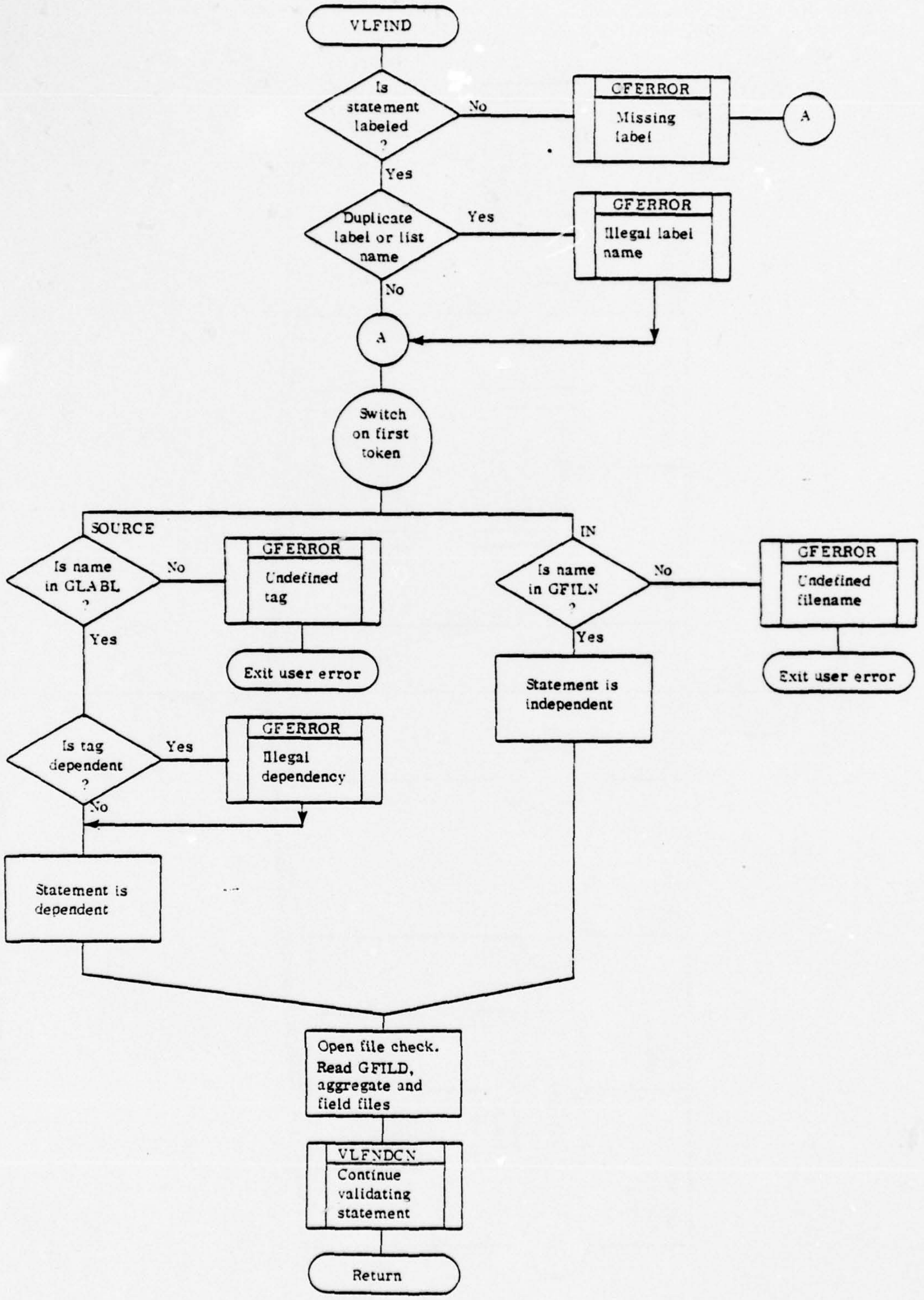


Figure 52. VALIDATE Process Data Flow (Sheet 6 of 13).

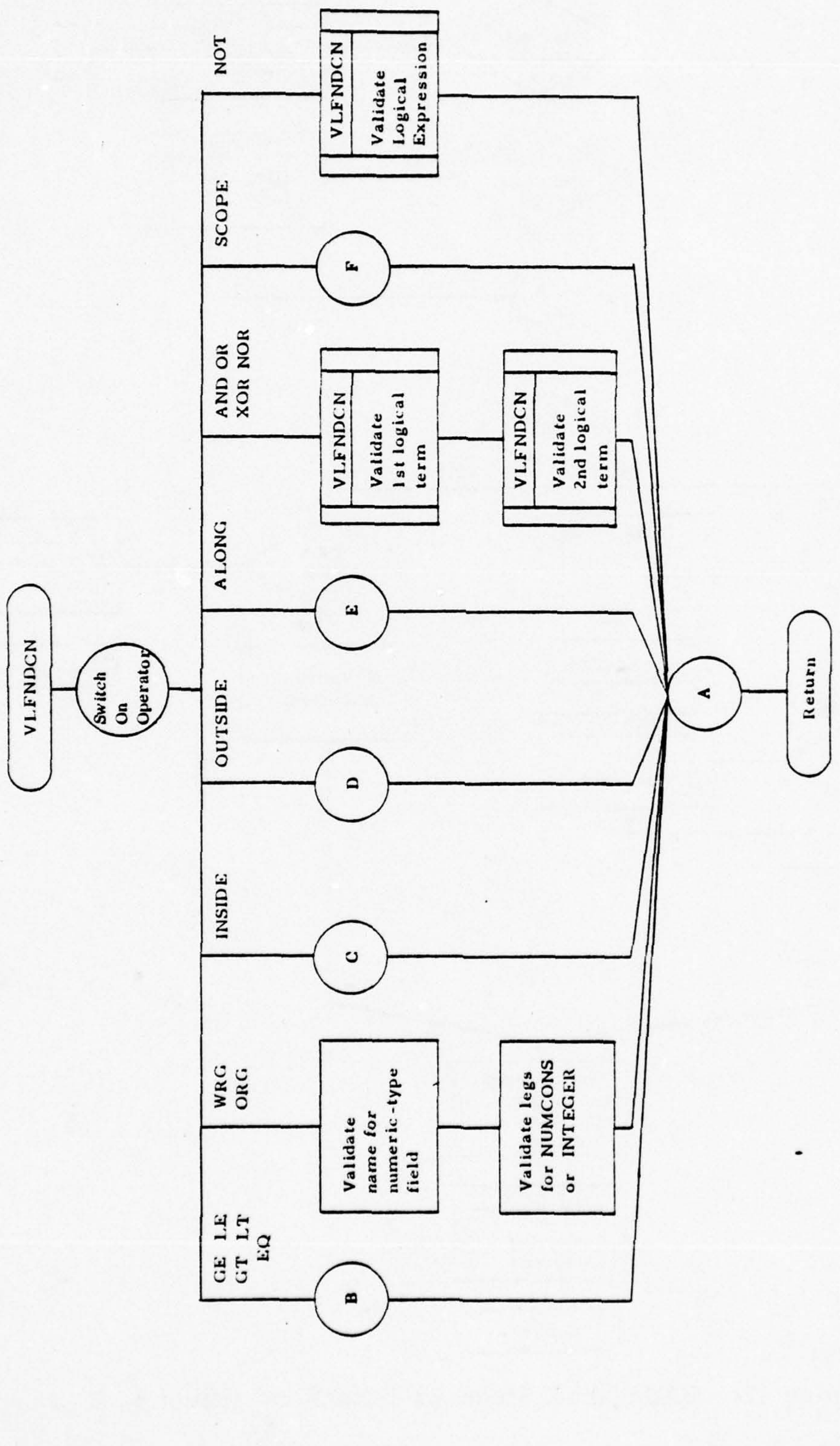

VLFNDCN
Switch On Operator
GE LE GT LT EQ
WRG ORG
INSIDE
OUTSIDE
ALONG
AND OR XOR NOR
SCOPE
NOT
B
Validate name for numeric-type field
Validate legs for NUMCONS or INTEGER
C
D
E
VLFNDCN
Validate 1st logical term
VLFNDCN
Validate 2nd logical term
F
VLFNDCN
Validate Logical Expression
A
Return

Figure 52. VALIDATE Process Data Flow (Sheet 7 of 13).

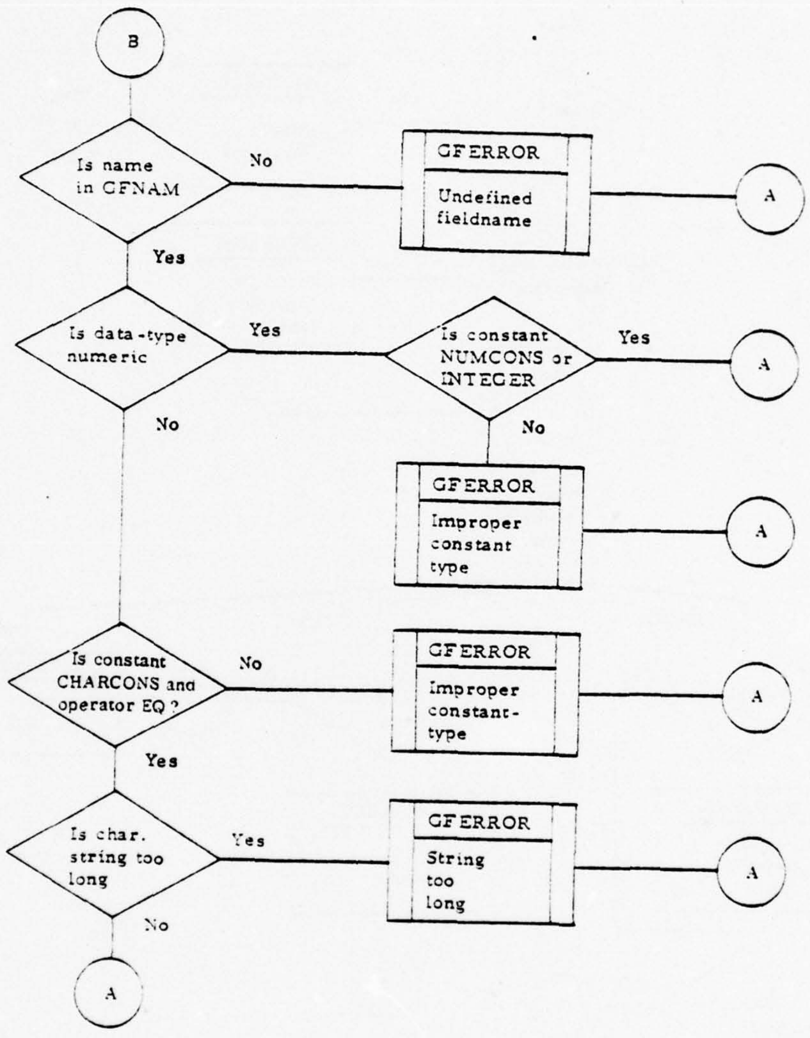


Figure 52. VALIDATE Process Data Flow (Sheet 8 of 13).

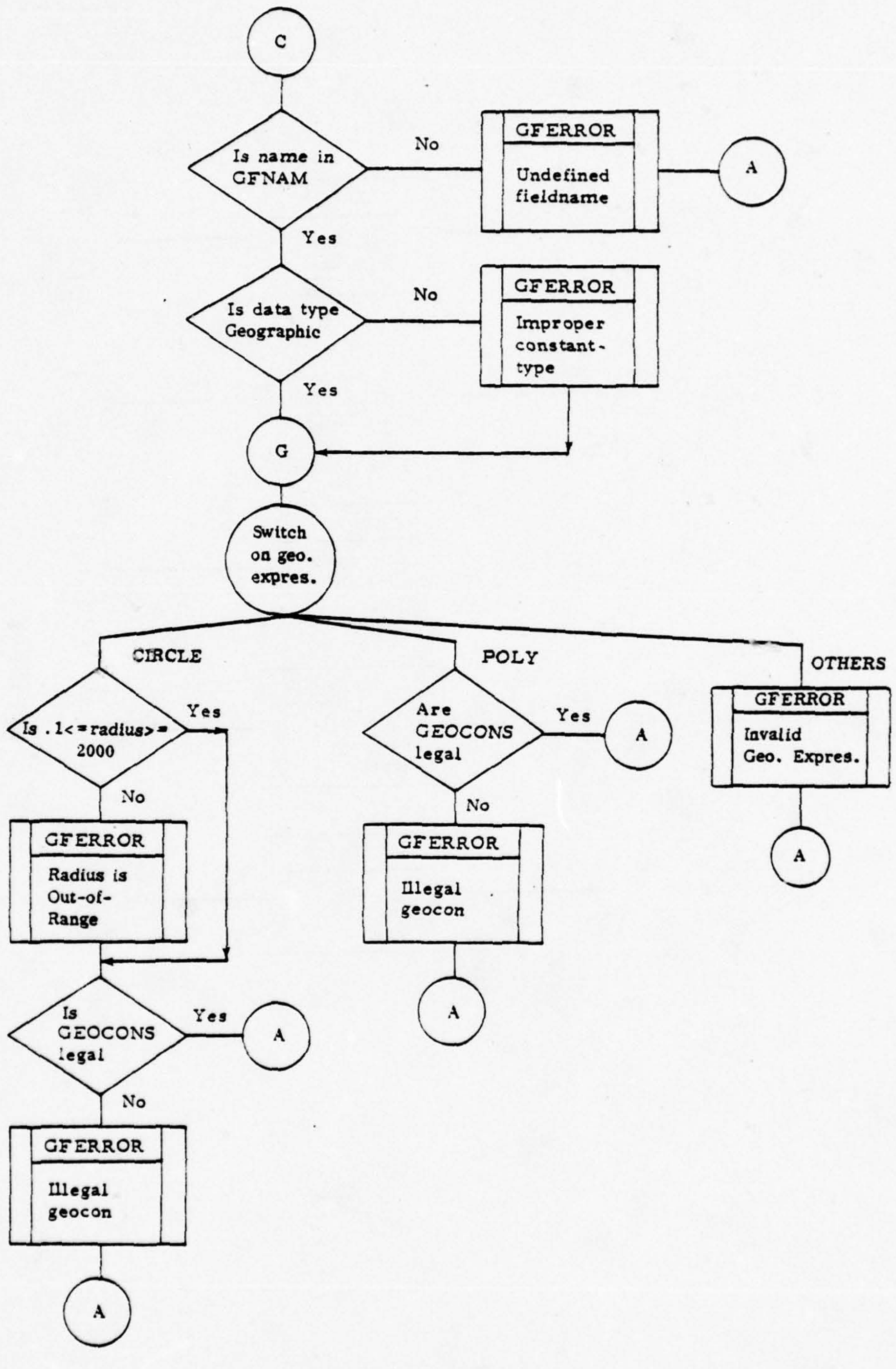


Figure 52. VALIDATE Process Data Flow (Sheet 9 of 13).

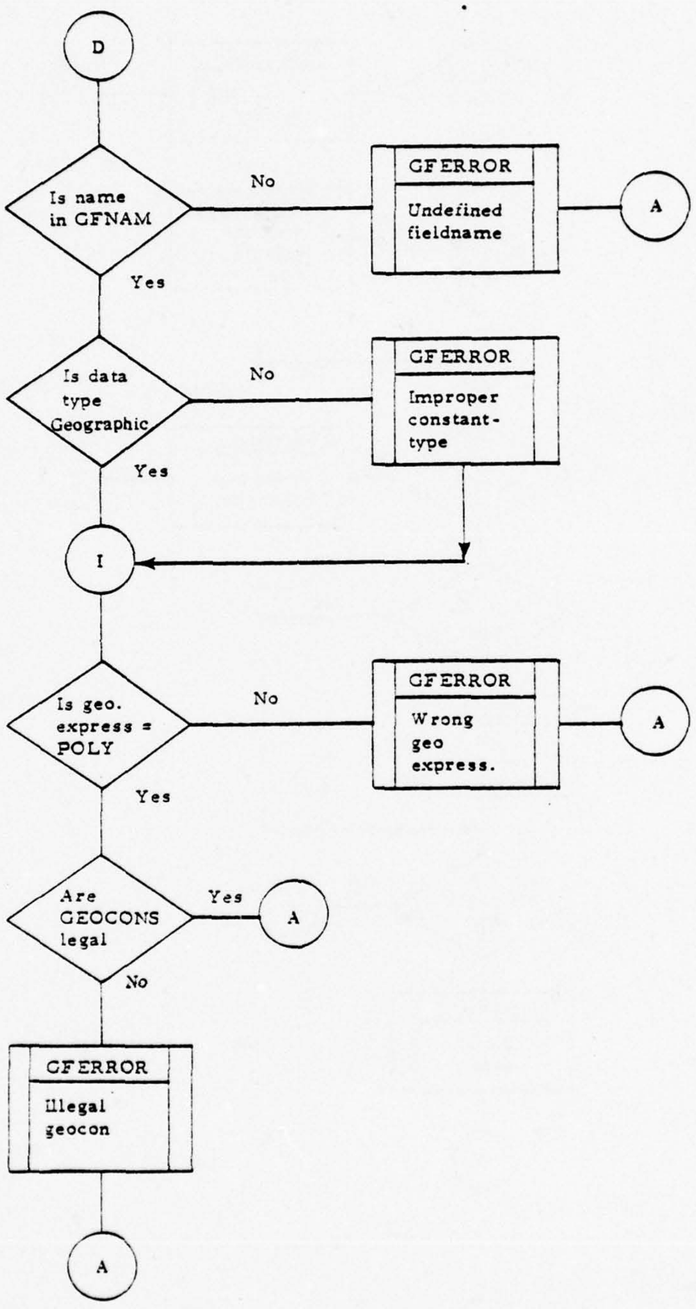


Figure 52. VALIDATE Process Data Flow (Sheet 10 of 13).

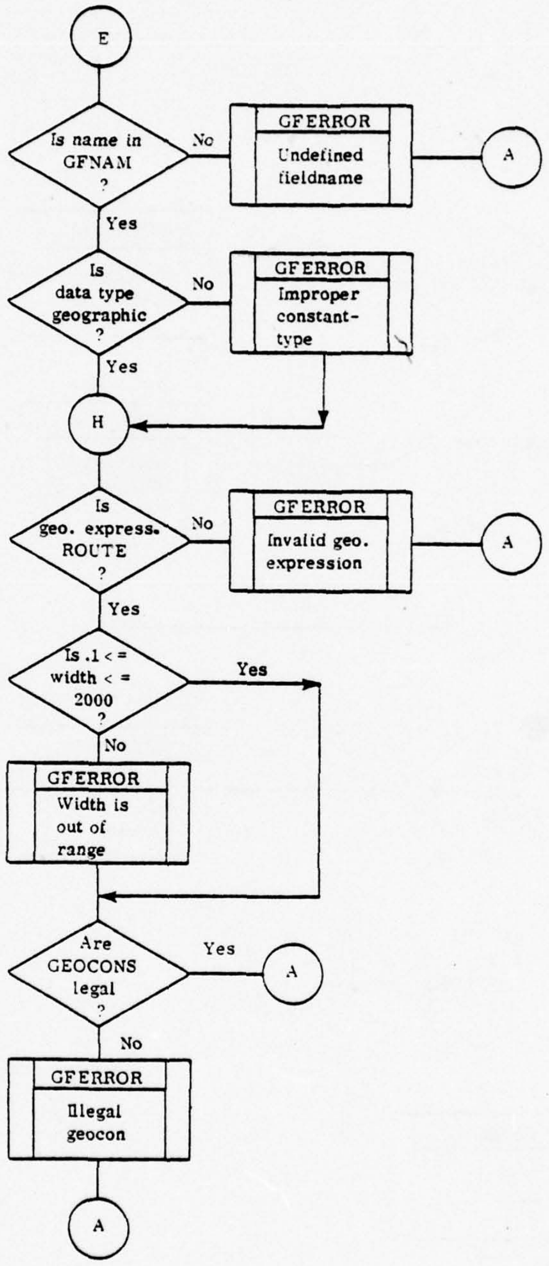


Figure 52. VALIDATE Process Data Flow (Sheet 11 of 13).

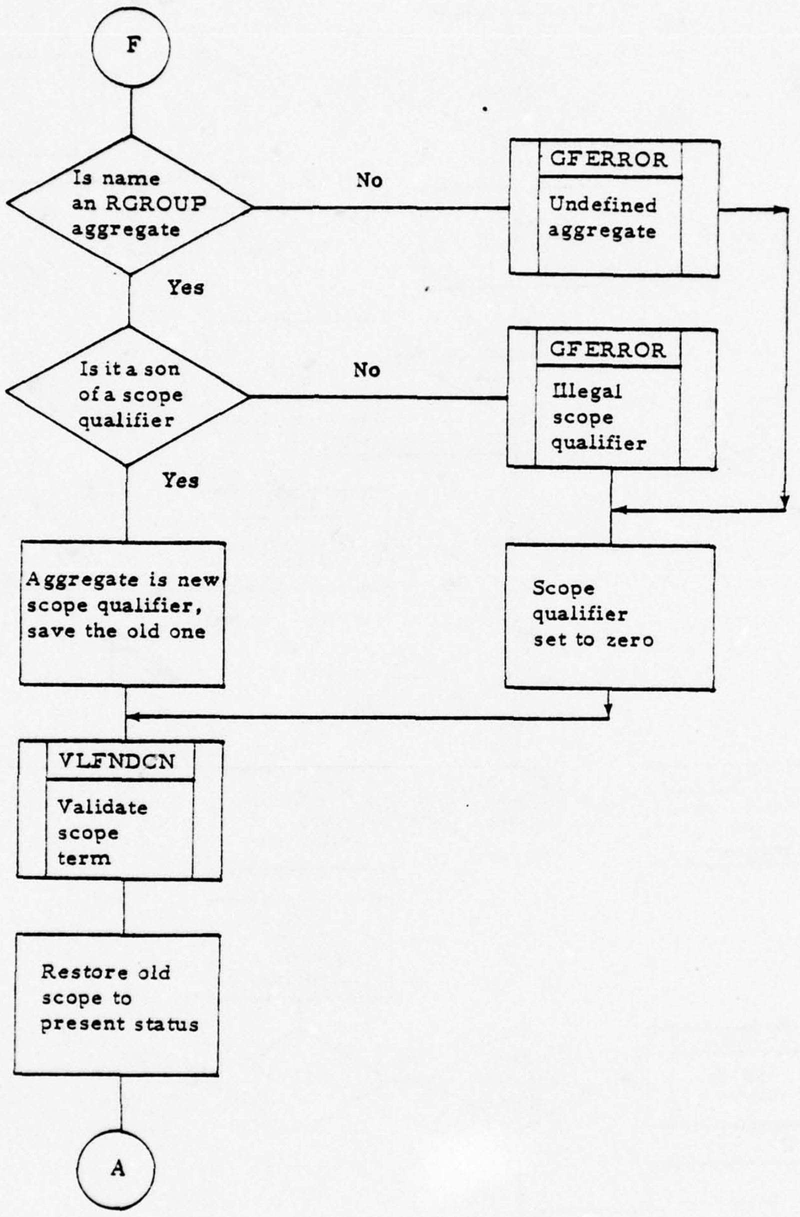

F
Is name an RGROUP aggregate
No
Yes
GFERROR
Undefined aggregate
Is it a son of a scope qualifier
No
Yes
GFERROR
Illegal scope qualifier
Aggregate is new scope qualifier, save the old one
Scope qualifier set to zero
VLFNDCN
Validate scope term
Restore old scope to present status
A

Figure 52. VALIDATE Process Data Flow (Sheet 12 of 13).

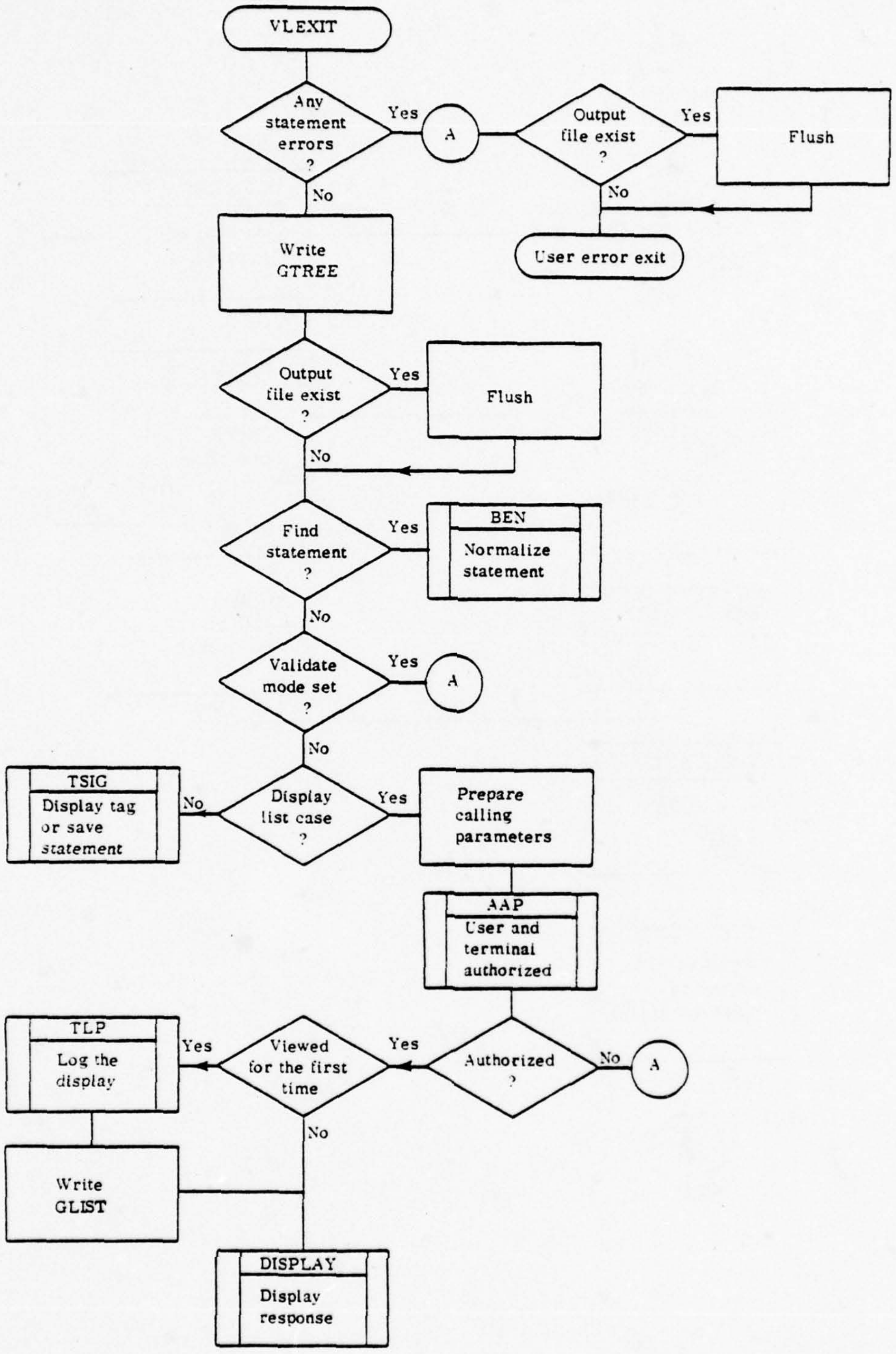


Figure 52. VALIDATE Process Data Flow (Sheet 13 of 13).

## VIEW

The VIEW process processes the UDL VIEW command. It displays list, label, file or transaction data as specified.

GLOBAL DATA USAGE – VIEW uses the following global data:

| | | |
|---|---|---|
| GLABL | – | label names and descriptions. |
| GTREE | – | parse tree. |
| GTDAT | – | parse tree data. |
| GFILN | – | file names. |
| GFILD | – | file descriptions. |
| GDBASE | – | database names. |
| GLIST | – | list names and descriptions. |
| GFNAM | – | field names. |
| GFDES | – | field descriptions. |
| GANAM | – | aggregate names. |
| GADES | – | aggregate descriptions. |
| GUSER | – | user descriptions. |
| GTDES | – | transaction descriptions. |
| GTFLN | – | transformation file names. |
| GTFLD | – | transformation file descriptions. |
| GTNAM | – | transformation field and aggregate names. |
| GTFDS | – | transformation field descriptions. |
| GTADS | – | transformation aggregate descriptions. |
| GPFDS | – | field position descriptions. |

LOCAL DATA USAGE – VIEW uses VILINE, a character array in which each print line is constructed.

GENERAL PROCESS FLOW – VIEW is activated whenever a user has input a VIEW command. VIEW first reads the parse tree (GTREE) and the file names (GFILN). If the command is a VIEW SCHEMA, VIEW TRANFILE, or VIEW TRANS with an argument, the parse tree data (GTDAT) is read. VIEW validates the file name for a VIEW SCHEMA or TRANFILE. If the file name is invalid, an error is output by VIERROR. When the file name is invalid, if VIEW SCHEMA was input, VIEW exits. If VIEW TRANFILE was input, then the user validation is made and if the user is the superuser, VIEW exits. If the user is not the superuser, VIEW outputs another error message and then exits.

If the command is VIEW SCHEMA, TRANFILE or FILE, the file descriptions file (GFILD) is read. VIEW then switches on the VIEW argument

If the command is VIEW LIST, VIEW reads the user's list descriptions (GLIST). If there are no lists, then VIERROR is called to output a NO LISTS message and exit. Otherwise, VIEW picks up each list index and examines that list's data. For each explicit (i.e., user generated) list, that list's name, status, and associated file name are output. Before the first explicit list's data are output, VIERROR is called to output a VIEW LIST header. If there are no explicit lists found, VIERROR is called to output a NO LISTS message and exit. Otherwise, after displaying the explicit lists' data, VIEW exits.

If the command is VIEW LABEL, VIEW reads the user's label descriptions (GLABL). If there are no labels, VIERROR is called to output a NO LABELS message and exit. Otherwise, VIERROR is called to output a VIEW LABEL header. For each label, VIEW outputs the label's name, associated file name, and dependent label name if applicable. After all labels have been processed, VIEW exits.

If the command is a VIEW SCHEMA, VIEW first reads in the data needed to describe a file's structure: field names and descriptions (GFNAM, GFDES) and aggregate names and descriptions (GANAM, GADES). It then outputs the VIEW SCHEMA header containing the file's name and its disposition; LOCAL, REMOTE-INTERACTIVE or REMOTE-BATCH. The field data for the basic data set are then output. For each field, the field's name, type, data type, interrogation attribute, display attribute, and size are displayed. For each aggregate, the aggregate's name and occurrence count (for arrays) are output. If any aggregate has subordinate aggregates, those subordinate aggregates and their field data are displayed. After all subordinate aggregates are displayed, any sibling aggregates and their subordinate aggregates are displayed. The file structure is presented in a hierarchical format. After the SCHEMA data are output, VIEW exits.

If the command is a VIEW TRANFILE, VIEW validates that the user is the superuser. If not, VIERROR is called to output an error and exit. VIEW reads the transformation file descriptions (GTFLD) and determines whether the file has a TRANFILE defined for it. If not, VIERROR is called to output an error and exit. VIEW then reads the data needed to describe a file's structure: GFDES, GFNAM, GADES and GANAM. Next, VIEW reads the transformation field descriptions (GTFDS) and, if the file has aggregates, the transformation aggregate descriptions (GTADS). If the file response data are position-oriented, VIEW reads the field position descriptions (GPFDS). Lastly, VIEW reads the transformation field and aggregate names (GTNAM) and the transformation file names (GTFLN). VIEW outputs a VIEW TRANFILE header containing the file's name, transformation name, host system (database) name, key field name if applicable, and number of characters per record if the file is position-oriented. The field data for the basic data set are then output. For each field, the field's name, transformation name, and, if the file is position-oriented, character position within a record and number of occurrences if applicable. For each aggregate, the aggregate's name and transformation name are output. If any aggregate has subordinate aggregates, those subordinate aggregates and their field data are displayed. After all subordinate aggregates are displayed, any sibling aggregates and their subordinate aggregates are displayed. The file structure is presented in a hierarchical format. After the TRANFILE data is output, VIEW exits.

If the command is a VIEW FILE, then VIEW checks to see whether there are any files defined. If not, VIERROR is called to output an error and exit. Otherwise, VIERROR is called to output the VIEW FILE header. For each file defined, VIEW outputs the file's name and disposition; LOCAL, REMOTE-BATCH, or REMOTE-INTERACTIVE. After the file data is output, VIEW exits.

If the command is a VIEW TRANS, VIEW reads the user descriptions (GUSER). If the user has no transactions, VIERROR is called to output an error and exit. VIEW then reads the first item of the transaction descriptions (GTDES), which contains the total transaction count for ADAPT. If no transaction number is specified, the following processing is performed. Each item in GTDES is read and examined to see if it is for this user. If it is, then it is checked to see if the user has already viewed it. If so, then it is ignored. Otherwise, the transaction's number, status, and associated file name are output. Before the first transaction's data are output, VIERROR is called to output a VIEW TRANS header. If there are no active transactions to output, then VIERROR is called to output a NO TRANS message and exit. After processing all transactions for the user, VIEW exits.

If a transaction number is specified, then VIEW checks to see if it is valid for the user. If not, VIERROR is called to output an error and exit. If the transaction status is not completed, VIERROR is called to output an error and exit. If the transaction status is answered, and the transaction was not initiated by a SAVE command, then the access authorization rights of the user and terminal must be verified. For each item in GTDES which relates to the transaction, VIEW performs a fork/execute sequence to the TAS AAP. The composite result of the authorization tests is saved. If the user can see the result, but the terminal is not authorized, then VIEW exits, If the user cannot see the answer, VITLP is called to cancel the batch jobs (if more than one), and VIDTRAN is called to delete the transaction and exit. If both the user and terminal can see the answer, then VITLP is called to mark the batch job(s) as delivered by calling the TAS TLP. Then the transaction's response, whether an answer or otherwise, is output to the user and VIDTRAN is called to delete the transaction and exit. See figure 53 for more information on process flow.

## MAJOR FUNCTION DESCRIPTIONS

VIDTRAN (Delete Transaction) – VIDTRAN locks GTDES via GFLOCK and then unlinks the transaction's response file. If the transaction was initiated by a SAVE command, the transaction is set to viewed by setting the list ID to minus one, GTDES is written out and VIERROR is called to exit. If the transaction is actually to be deleted, each item in GTDES which relates to the transaction is read in, set to zeroes, and written back to disk. The first item of GTDES is read, the transaction count is decremented, and the item is written back out. GTDES is unlocked via GFUNLOCK and GUSER is locked via GFLOCK. VIDTRAN reads GUSER, decrements the user's transaction count, and writes GUSER back out. VIERROR is called to output a final message and exit.

<u>VITLP (Call TAS Logging Process)</u> – VITLP is called with a function code of either cancel or delivered. VITLP performs a fork/execute sequence to the TAS TLP for each item in GTDES which relates to the current transaction. After all GTDES items have been processed, VITLP returns.

<u>VIERROR (Output Error/Message and/or Exit)</u> – VIERROR checks to see if a message number has been specified. If so, an error or message is output via GFERROR. VIERROR next checks to see if the process is to exit. If not, VIERROR returns. Otherwise, VIERROR determines if any files are locked, and, if so, unlocks them via GFUNLOCK. VIERROR then exits.

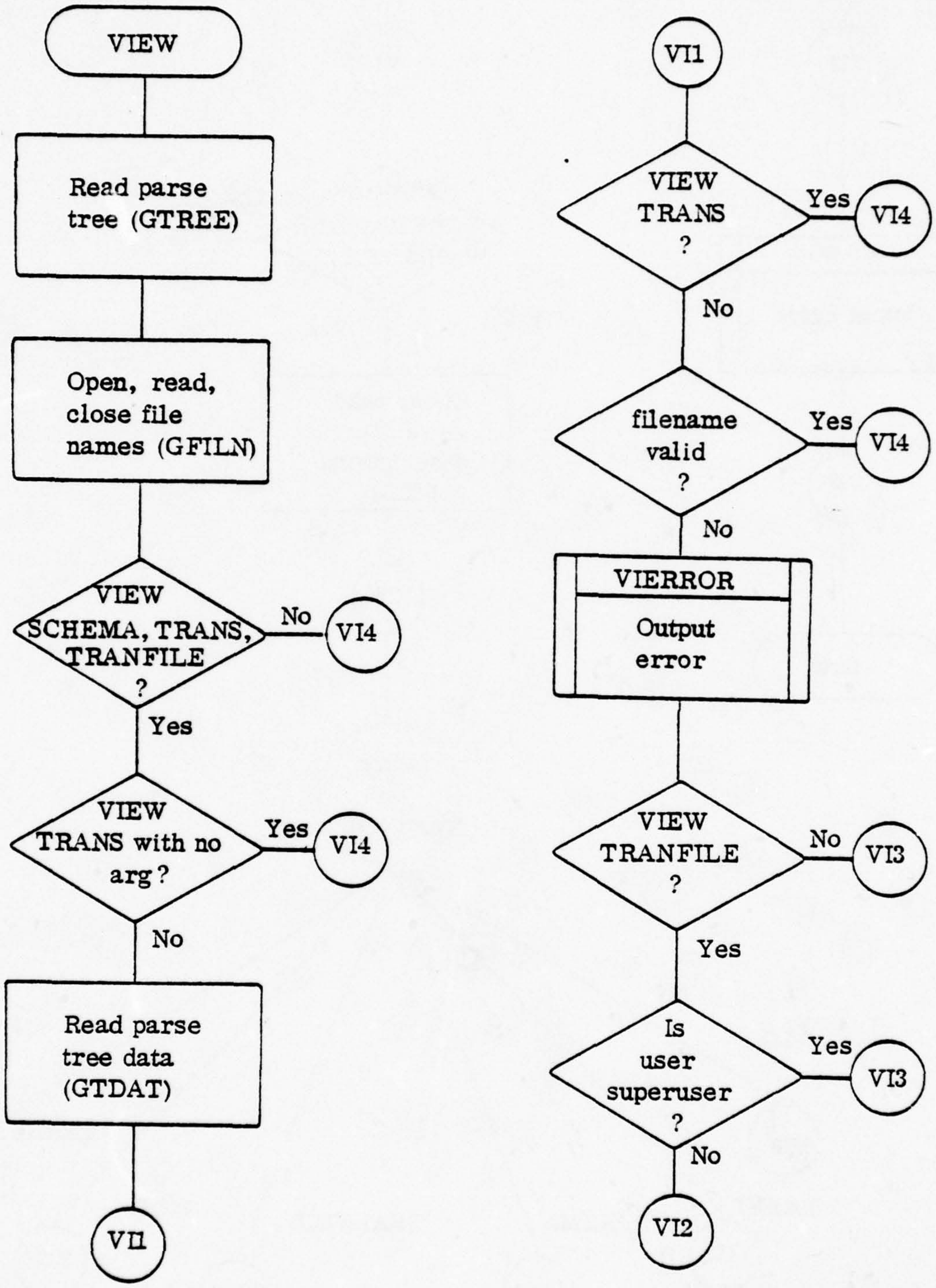


Figure 53. VIEW Process Data Flow (Sheet 1 of 16).

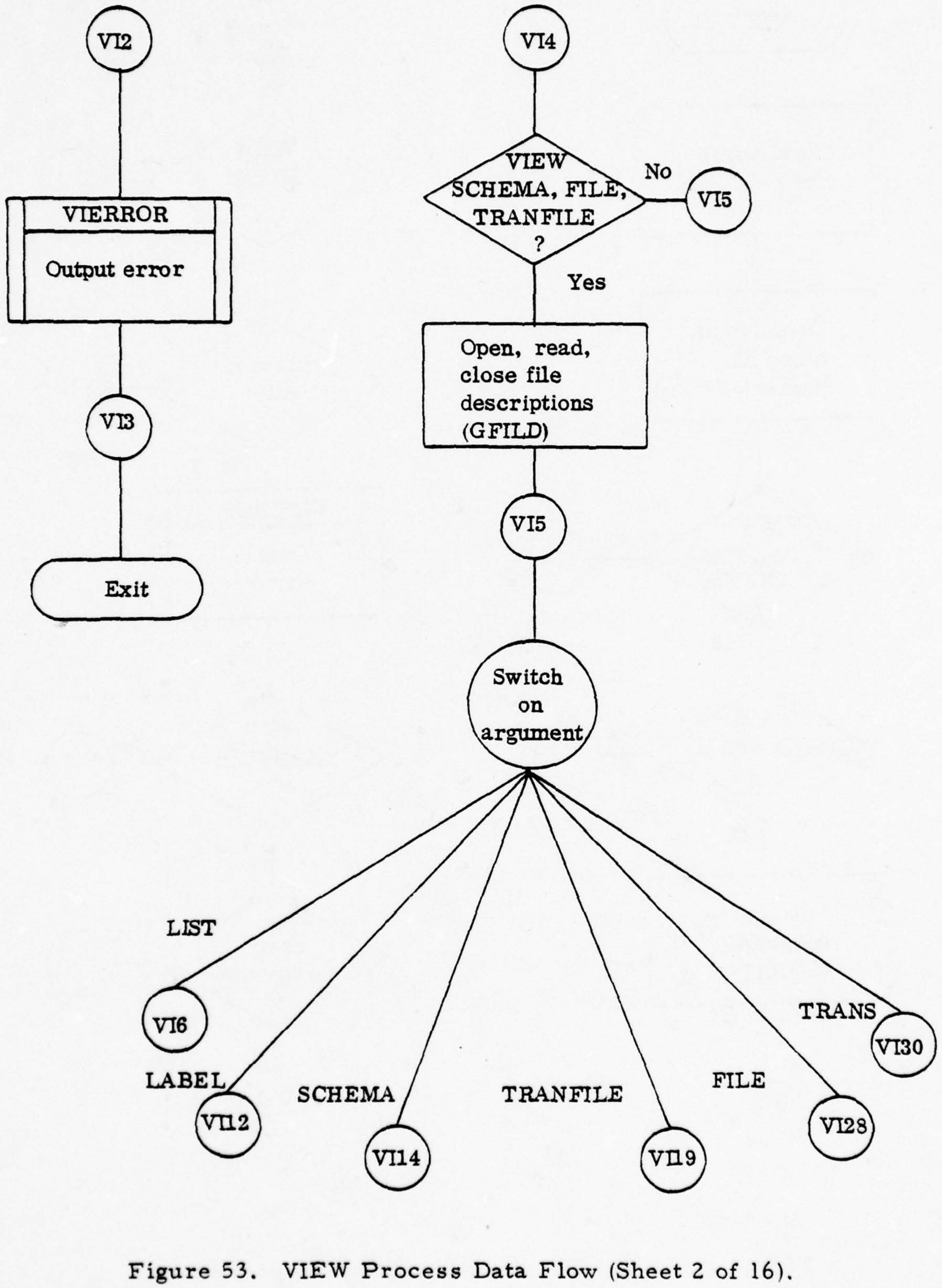


Figure 53. VIEW Process Data Flow (Sheet 2 of 16).

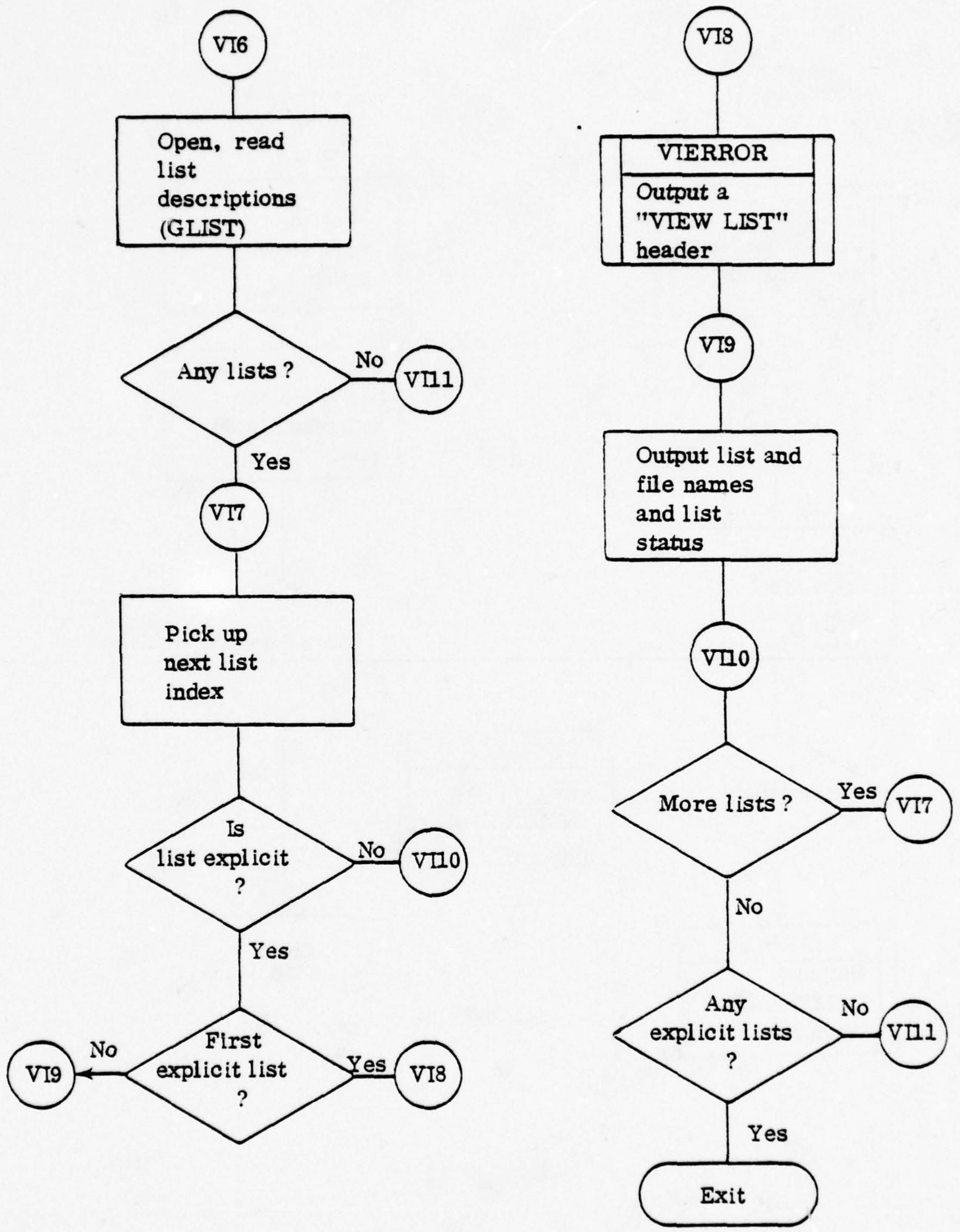


Figure 53. VIEW Process Data Flow (Sheet 3 of 16).

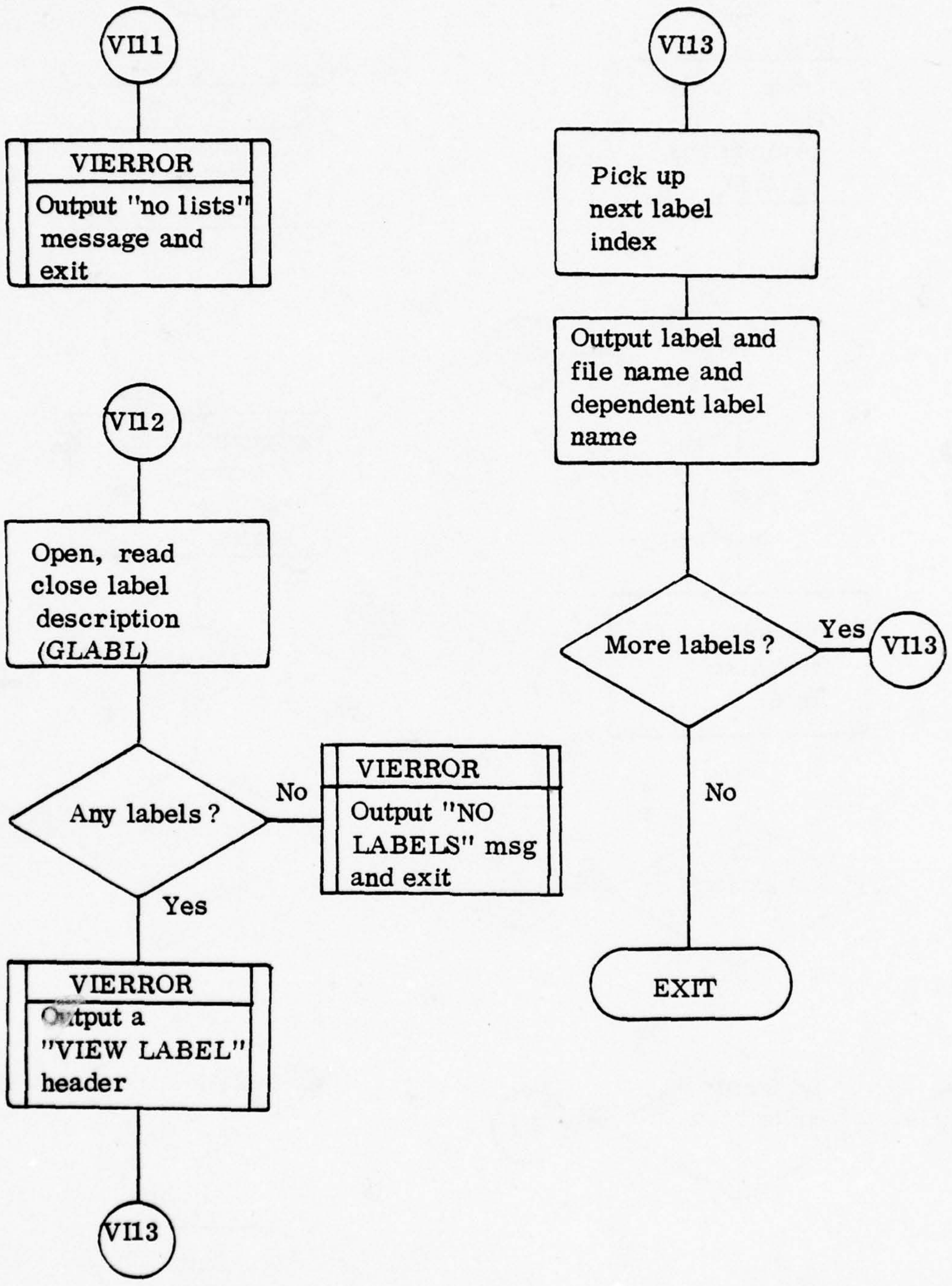
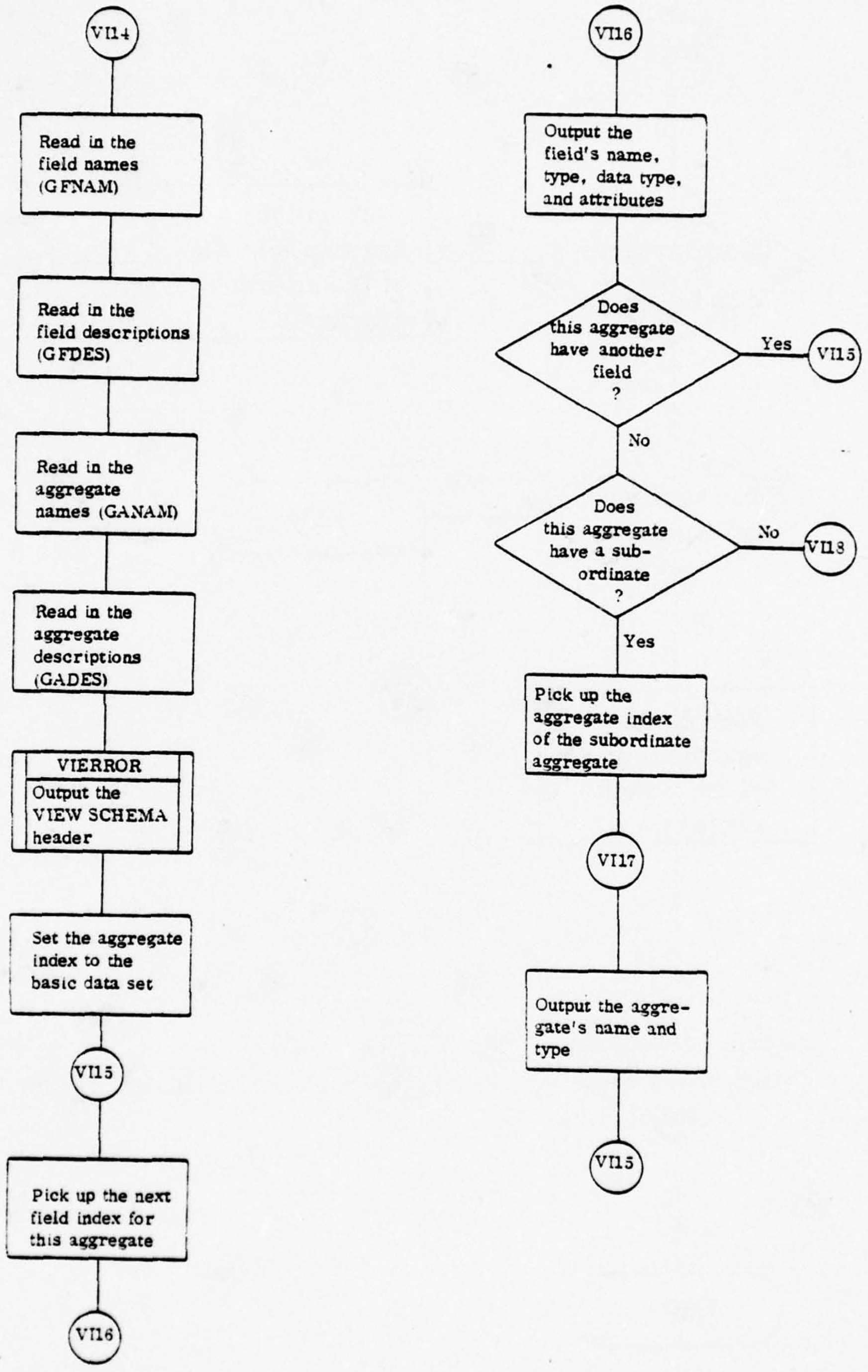

VI11
VIERROR
Output "no lists" message and exit
VI12
Open, read close label description (GLABL)
Any labels ?
No
VIERROR
Output "NO LABELS" msg and exit
Yes
VIERROR
Output a "VIEW LABEL" header
VI13
VI13
Pick up next label index
Output label and file name and dependent label name
More labels ?
Yes
VI13
No
EXIT

Figure 53. VIEW Process Data Flow (Sheet 4 of 16).



VI14
Read in the field names (GFNAM)
Read in the field descriptions (GFDES)
Read in the aggregate names (GANAM)
Read in the aggregate descriptions (GADES)
VIERROR
Output the VIEW SCHEMA header
Set the aggregate index to the basic data set
VI15
Pick up the next field index for this aggregate
VI16
VI16
Output the field's name, type, data type, and attributes
Does this aggregate have another field ?
Yes
VI15
No
Does this aggregate have a sub-ordinate ?
No
VI18
Yes
Pick up the aggregate index of the subordinate aggregate
VI17
Output the aggre-gate's name and type
VI15

Figure 53. VIEW Process Data Flow (Sheet 5 of 16).

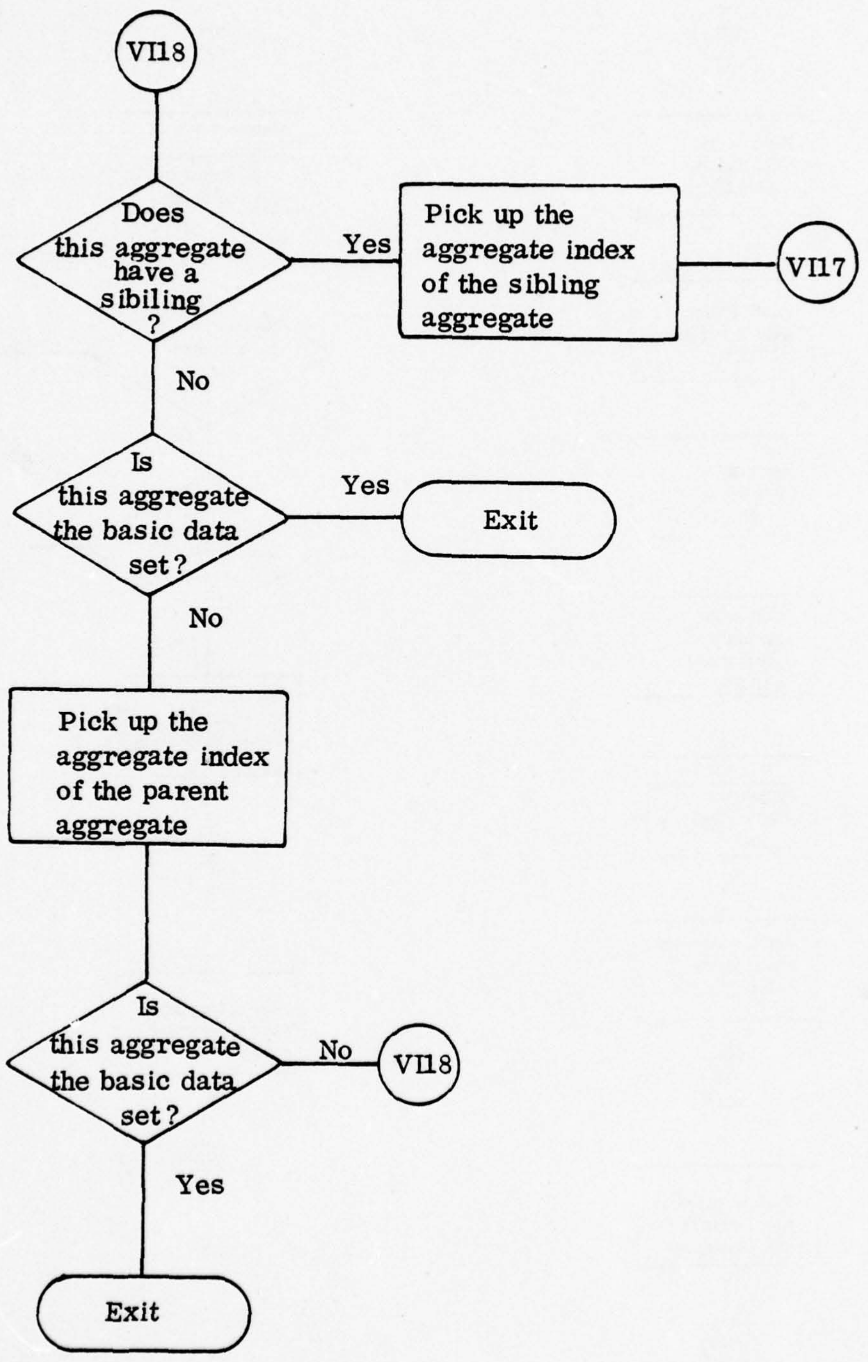

VI18
Does this aggregate have a sibling ?
Yes
Pick up the aggregate index of the sibling aggregate
VI17
No
Is this aggregate the basic data set ?
Yes
Exit
No
Pick up the aggregate index of the parent aggregate
Is this aggregate the basic data set ?
No
VI18
Yes
Exit

Figure 53. VIEW Process Data Flow (Sheet 6 of 16).

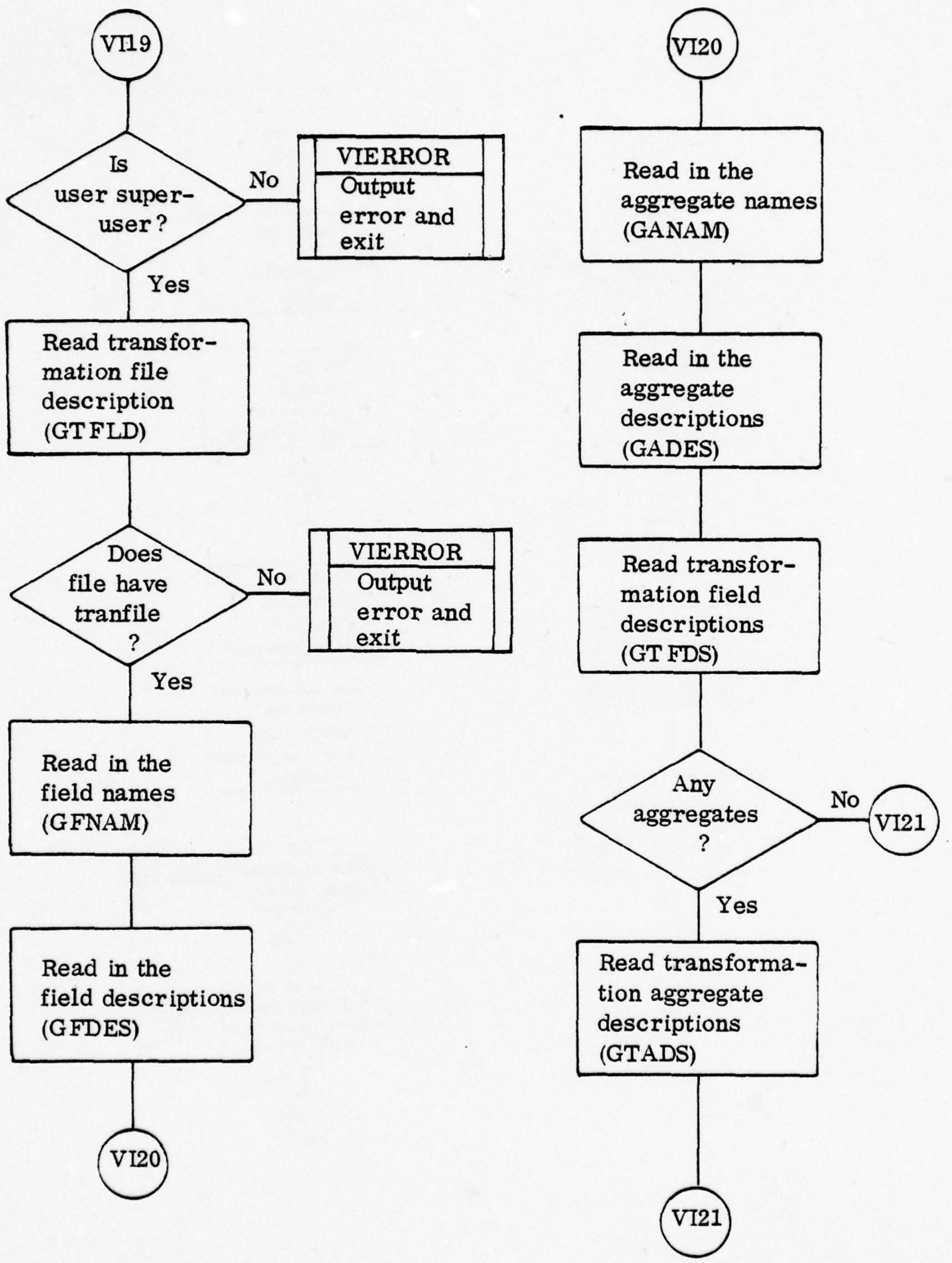

VI19
Is user super-user?
No
VIERROR
Output error and exit
Yes
Read transformation file description (GTFLD)
Does file have tranfile ?
No
VIERROR
Output error and exit
Yes
Read in the field names (GFNAM)
Read in the field descriptions (GFDES)
VI20
VI20
Read in the aggregate names (GANAM)
Read in the aggregate descriptions (GADES)
Read transformation field descriptions (GT FDS)
Any aggregates ?
No
VI21
Yes
Read transformation aggregate descriptions (GTADS)
VI21

Figure 53. VIEW Process Data Flow (Sheet 7 of 16).

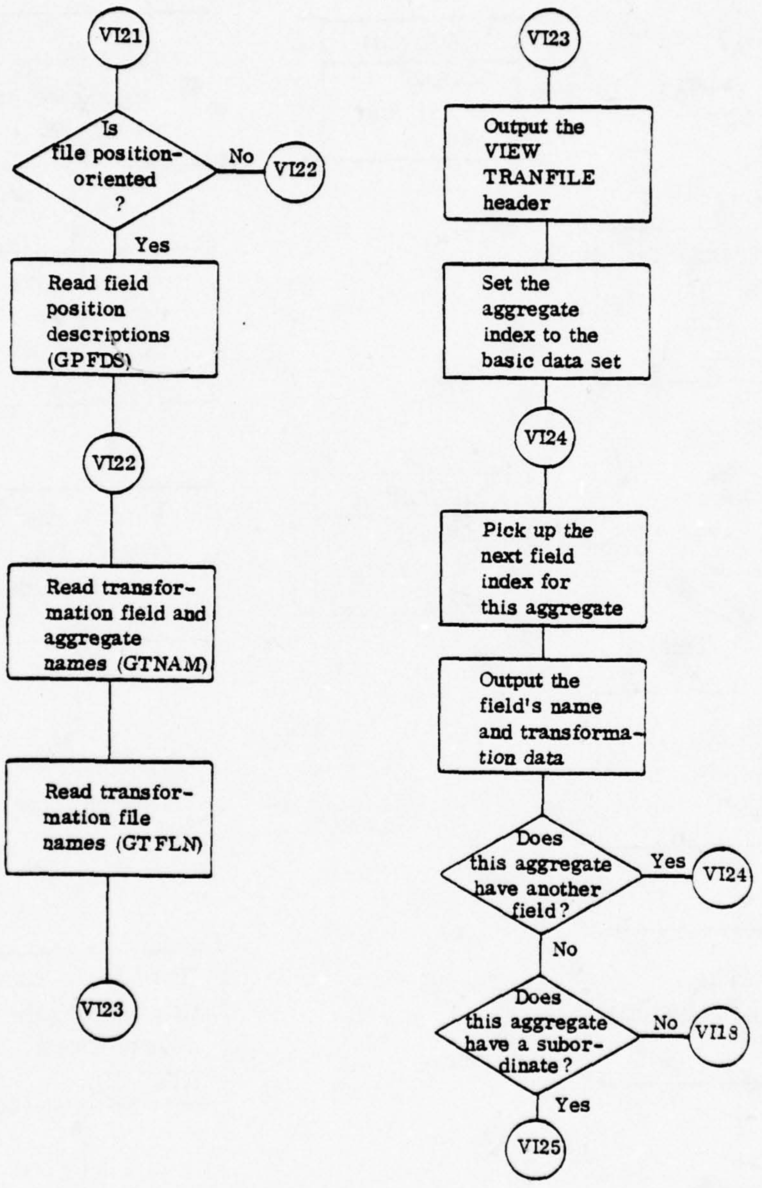

VI21
Is file position-oriented ?
No
VI22
Yes
Read field position descriptions (GPFDS)
VI22
Read transformation field and aggregate names (GTNAM)
Read transformation file names (GTFLN)
VI23
VI23
Output the VIEW TRANFILE header
Set the aggregate index to the basic data set
VI24
Pick up the next field index for this aggregate
Output the field's name and transformation data
Does this aggregate have another field ?
Yes
VI24
No
Does this aggregate have a subordinate ?
No
VI18
Yes
VI25

Figure 53. VIEW Process Data Flow (Sheet 8 of 16).

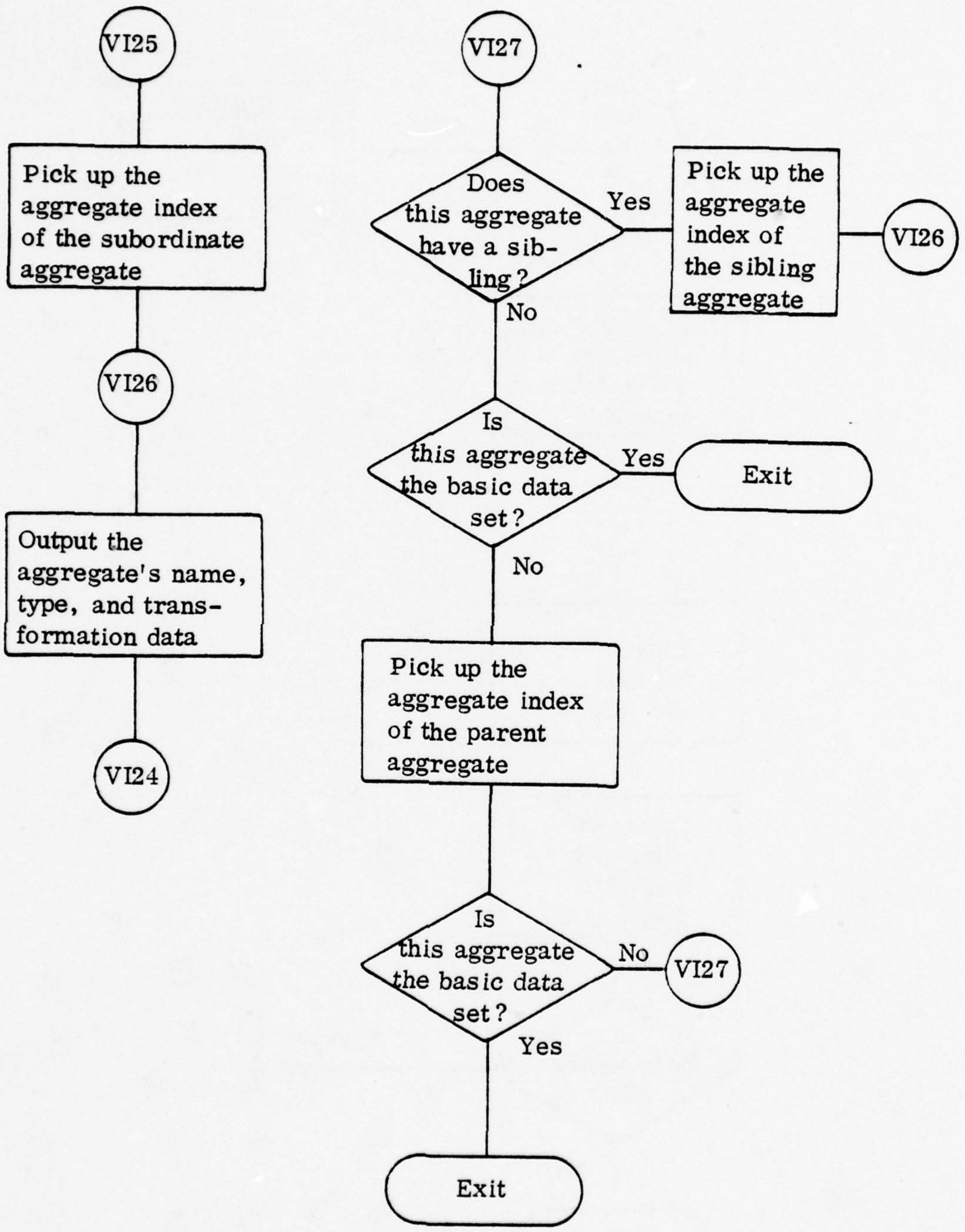

VI25
Pick up the aggregate index of the subordinate aggregate
VI26
Output the aggregate's name, type, and transformation data
VI24
VI27
Does this aggregate have a sibling?
Yes
Pick up the aggregate index of the sibling aggregate
VI26
No
Is this aggregate the basic data set?
Yes
Exit
No
Pick up the aggregate index of the parent aggregate
Is this aggregate the basic data set?
No
VI27
Yes
Exit

Figure 53. VIEW Process Data Flow (Sheet 9 of 16).

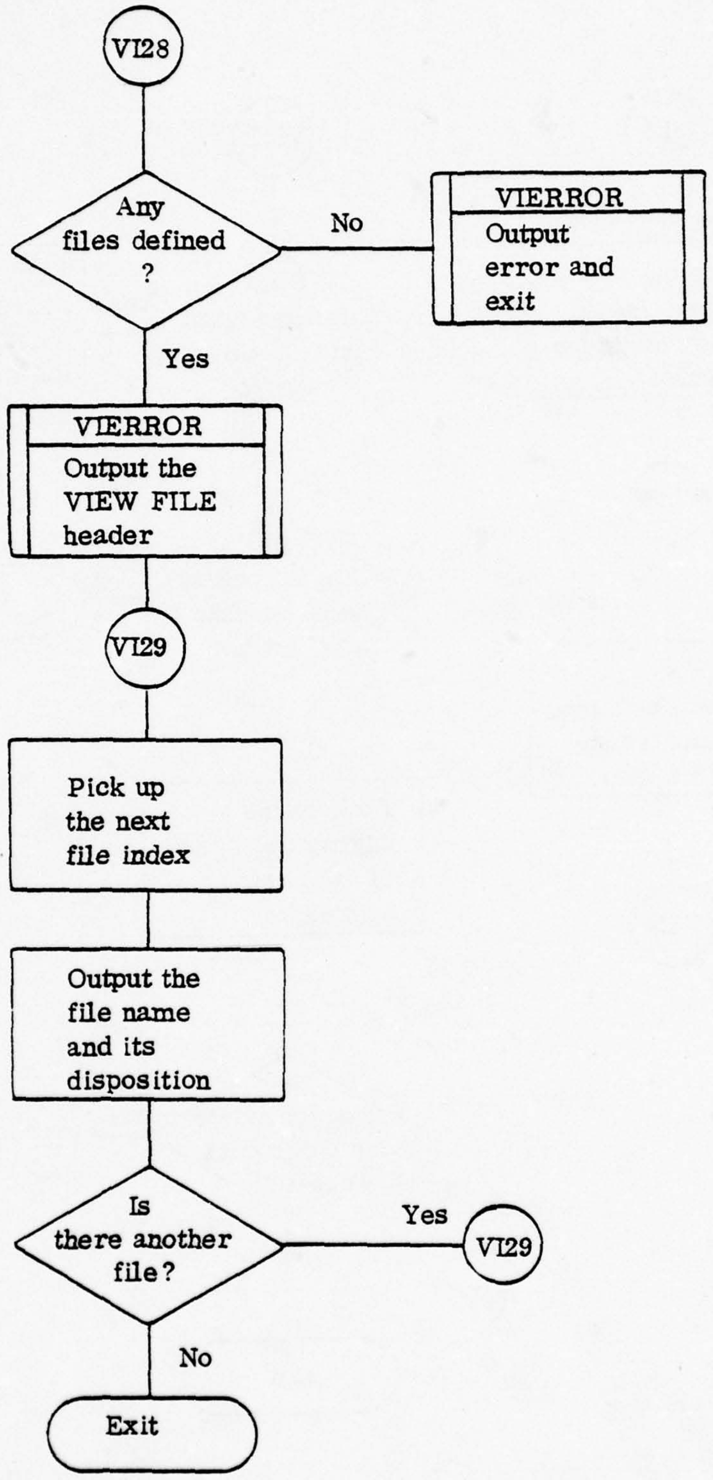


Figure 53. VIEW Process Data Flow (Sheet 10 of 16).

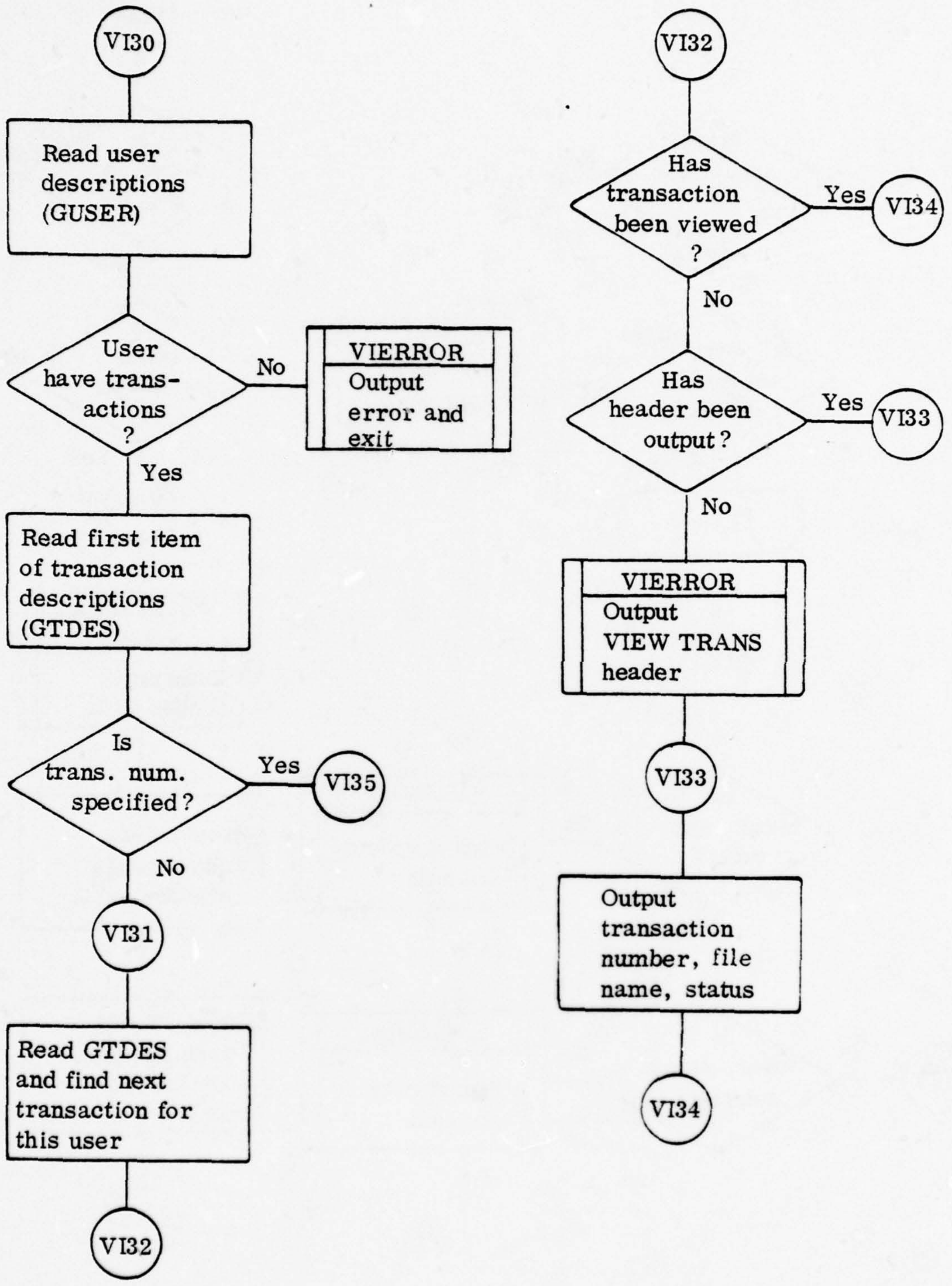

VI30
Read user descriptions (GUSER)
User have trans-actions ?
No
VIERROR
Output error and exit
Yes
Read first item of transaction descriptions (GTDES)
Is trans. num. specified?
Yes
VI35
No
VI31
Read GTDES and find next transaction for this user
VI32
VI32
Has transaction been viewed ?
Yes
VI34
No
Has header been output?
Yes
VI33
No
VIERROR
Output VIEW TRANS header
VI33
Output transaction number, file name, status
VI34

Figure 53. VIEW Process Data Flow (Sheet 11 of 16).

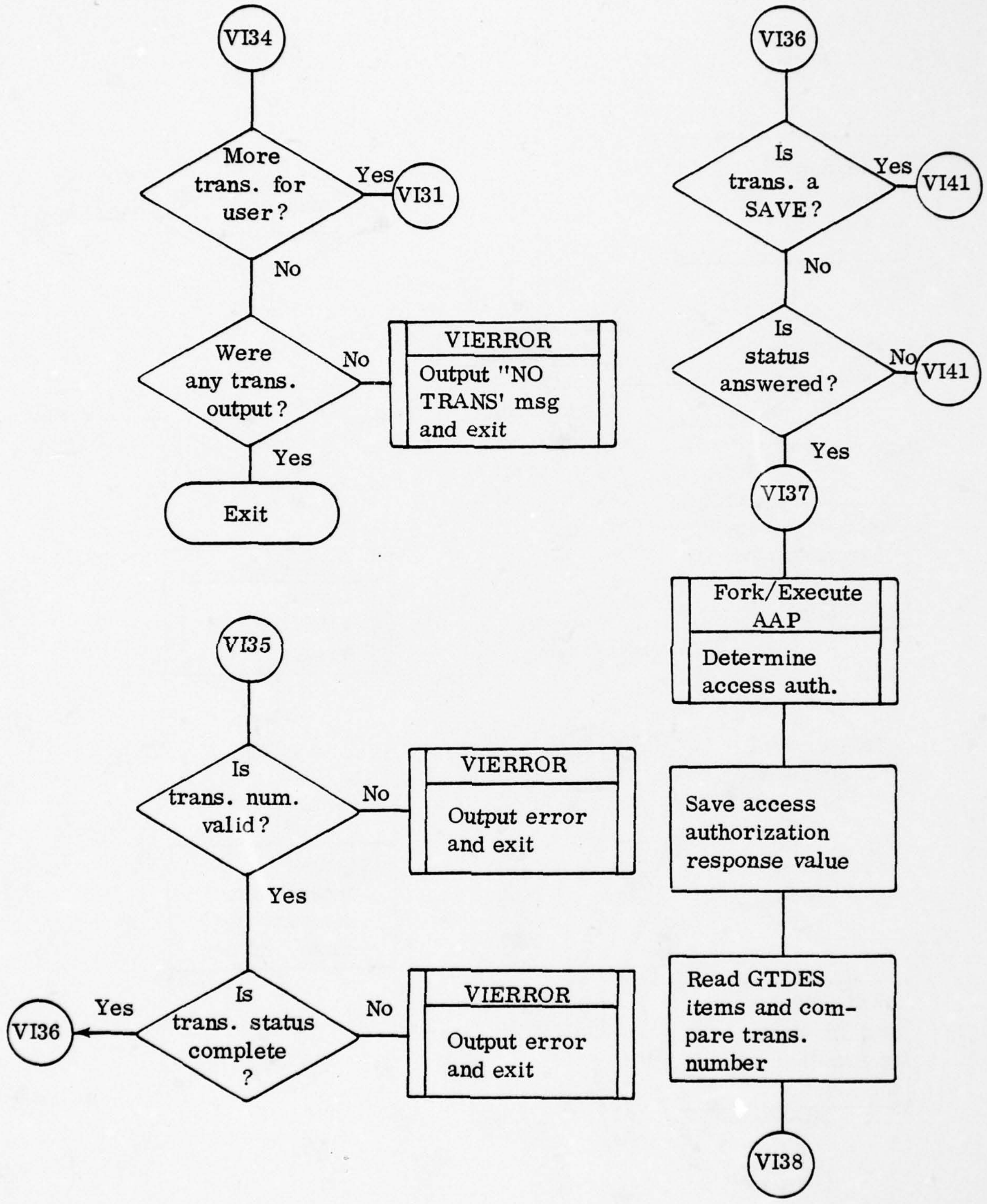


Figure 53. VIEW Process Data Flow (Sheet 12 of 16).

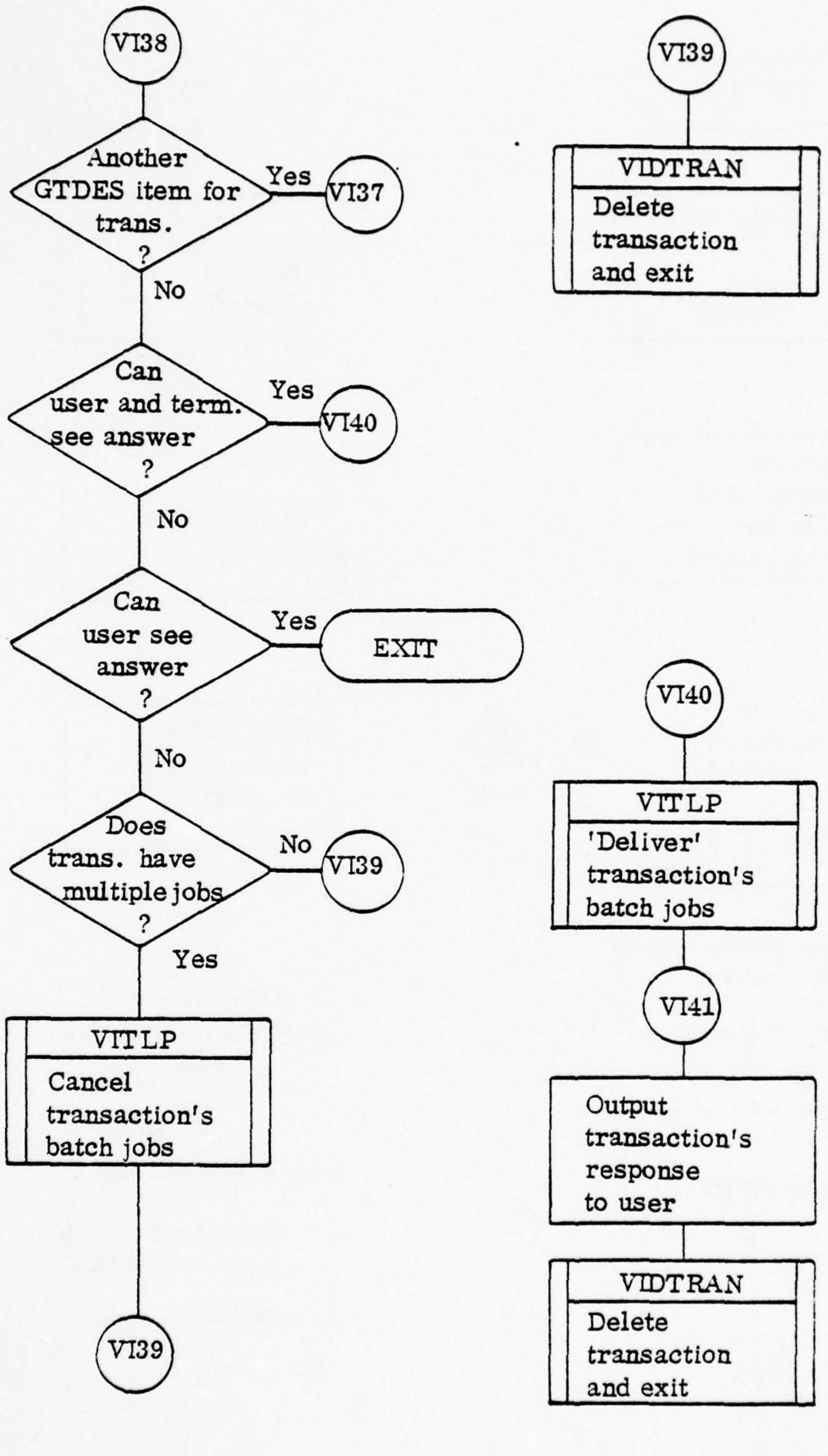
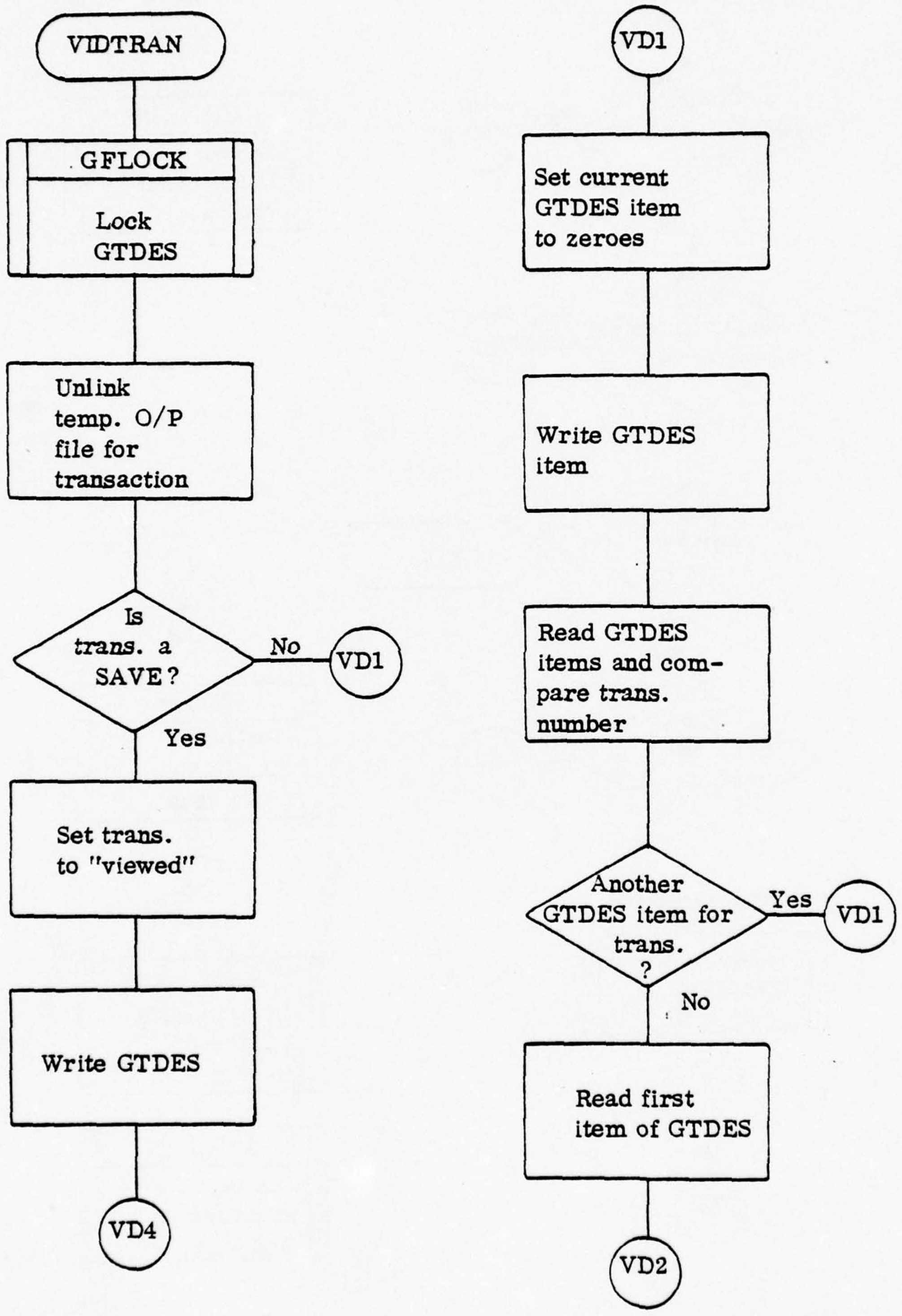

VI38
Another GTDES item for trans. ?
Yes
VI37
No
Can user and term. see answer ?
Yes
VI40
No
Can user see answer ?
Yes
EXIT
No
Does trans. have multiple jobs ?
No
VI39
Yes
VITLP
Cancel transaction's batch jobs
VI39
VI39
VIDTRAN
Delete transaction and exit
VI40
VITLP
'Deliver' transaction's batch jobs
VI41
Output transaction's response to user
VIDTRAN
Delete transaction and exit

Figure 53. VIEW Process Data Flow (Sheet 13 of 16).



VIDTRAN
GFLOCK
Lock GTDES
Unlink temp. O/P file for transaction
Is trans. a SAVE?
No
VD1
Yes
Set trans. to "viewed"
Write GTDES
VD4
VD1
Set current GTDES item to zeroes
Write GTDES item
Read GTDES items and compare trans. number
Another GTDES item for trans. ?
Yes
VD1
No
Read first item of GTDES
VD2

Figure 53. VIEW Process Data Flow (Sheet 14 of 16).

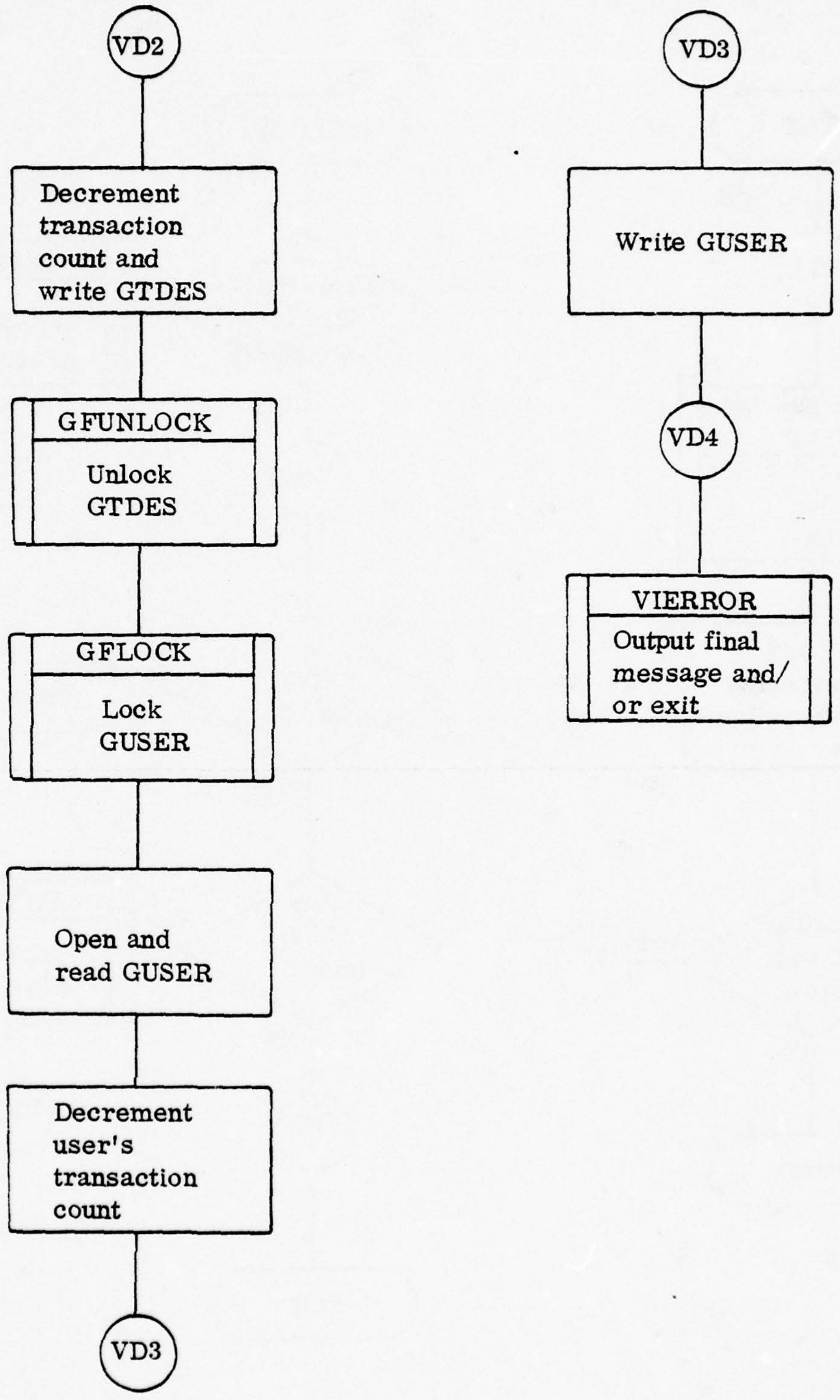

VD2
Decrement transaction count and write GTDES
GFUNLOCK
Unlock GTDES
GFLOCK
Lock GUSER
Open and read GUSER
Decrement user's transaction count
VD3
VD3
Write GUSER
VD4
VIERROR
Output final message and/ or exit

Figure 53. VIEW Process Data Flow (Sheet 15 of 16).

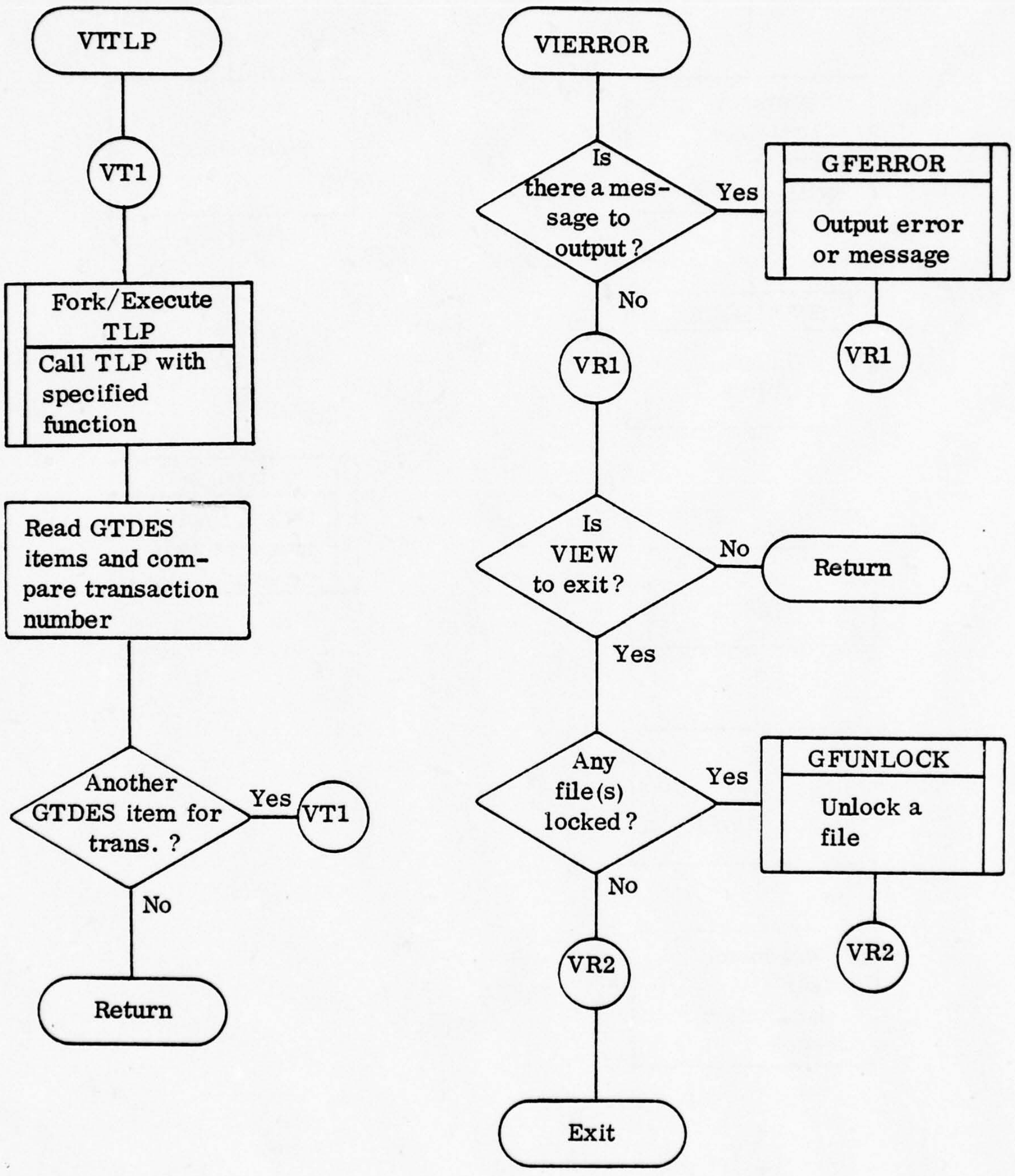


Figure 53. VIEW Process Data Flow (Sheet 16 of 16).

## ADAPT I GLOBAL FUNCTIONS AND DATA

This section provides descriptions of the global functions and global data utilized by ADAPT I. Global functions and data are defined as those functions and data which are commonly utilized by more than one process of ADAPT I. Functions are defined in the literal sense as in the C language, hence are not processes but actual functions which are compiled with the individual processes that call them. Global data are different in this respect since only a single copy of these data actually exists in the ADAPT I environment.

This section is divided into two subsections, one describing global functions and the other providing descriptions of the many global data structures and arrays existing in ADAPT I. Currently only one global function has been defined for ADAPT I. Three global functions defined for TAS are also utilized by ADAPT and are described herein. Twenty-seven global data structures are currently defined for ADAPT I.

GLOBAL FUNCTIONS – ADAPT I currently requires the use of four global functions:

a. GFERROR – outputs general diagnostic messages.

b. GFLOCK – locks read-for-update data files (TAS).

c. GFUNLOCK – unlocks read-for-update data files (TAS).

d. GFJOBID – assigns TAS unique JOBIDs for batch transactions.

Following are the descriptions of these functions.

## GFERROR

Input Parameters – GFERROR requires two input parameters:

a. An ADAPT I error message number (integer).

b. A parameter which is either zero or a pointer to a null-terminated character string.

Output Parameters – None.

Operation – GFERROR provides ADAPT I with a common means for the output of diagnostic and system error messages to the user. The first input parameter specifies a canned message and the second parameter provides a character string which is to be inserted into an appropriate place in the canned message, if applicable. If the message is not a system error, the error message number indicates the appropriate error description item in GFEDES (error/message descriptions) which, in turn, contains pointers to the canned message stored in GFEMES. The character string pointed to by the second input parameter (if non-zero), is stored in the message and output to the user's terminal. GFERROR then returns to the calling function. If the message is a system error, the TAS operator's console is locked via GFLOCK and the message (as stored in GFSYSERR) is output to the console. Once the message has been printed, the operator's console is unlocked via GFUNLOCK. The user is then informed that a system error has occurred while his request was processed and GFERROR returns to the calling function.

## GFLOCK

Input Parameters – GFLOCK requires one input parameter: a pathname specifying a global data file.

Output Parameters – None.

Operation – It is necessary to protect the integrity of certain ADAPT I global data during update sequences, i. e.; for read-for-update protection on a given file. That is, only one process may read-for-update a given file at any one time. The TAS global function GFLOCK (and its counterpart, GFUNLOCK) provides this protection within the TAS and ADAPT I environments.

GFLOCK uses the input parameter, a global data file pathname, to lock out the file during a read-for-update sequence. The character string "-lock" is appended to the file pathname to generate the lock-file pathname. GFLOCK then attempts to create the lock-file with the read-only mode. If this file already exists, an error status is received and the function sleeps for one second, and then attempts the create again. Eventually, the create is successful and GFLOCK returns to the calling function. At this point, the specified global data file is locked out from other read-for-update attempts. Of course, this assumes the cooperation of other processes to call GFLOCK prior to attempting to read-for-update a file. The file can now be read in, updated, and then written back out. After the updating process is completed, it is mandatory that GFUNLOCK be called to unlock the global data file.

## GFUNLOCK

Input Parameters – GFUNLOCK requires one input parameter: a pathname specifying a global data file.

Output Parameters – None.

Operation – GFUNLOCK is a counterpart to GFLOCK and must be called when a given locked global file has been updated and written out.

Basically, GFUNLOCK appends the character string "-lock" to the file pathname and unlinks the associated lock-file thus allowing another process to create it, hence gaining control of the file. Failure to call GFUNLOCK after successfully locking out the file with GFLOCK would eventually bring the ADAPT I system to a halt.

## GFJOBID

Input Parameter – GFJOBID has one input parameter: the address of a six-character storage buffer.

Output Parameter – GFJOBID returns the status of zero if the process was successful or an error status if a system error occurred, i.e., read or write error.

Operations – When called, GFJOBID first saves the interrupt signal state and ignores further interrupts. GJOBID is then opened, locked (via GFLOCK) and read. The contents of GJOBID are copied into the user buffer pointed to by the argument to GFJOBID. The JOBID counter is then incremented and rewritten. GJOBID is then closed and unlocked (via GFUNLOCK). The interrupt signal status is reset and GFJOBID returns with the status of any system errors which may have occurred during processing or a zero if none occurred.

GLOBAL DATA – Currently, twenty-seven global data structures have been defined for ADAPT I. They are as follows:

| | | | |
|---|---|---|---|
| a. | GADES | – | aggregate descriptions. |
| b. | GANAM | – | aggregate names. |
| c. | GFDES | – | field descriptions. |
| d. | GFNAM | – | field names. |
| e. | GFILD | – | file descriptions. |
| f. | GFILN | – | file names. |
| g. | GRMAP | – | internal record map. |
| h. | GMPHD | – | internal record map header. |
| i. | GRDAT | – | internal record data. |
| j. | GTREE | – | parse tree. |
| k. | GTDAT | – | parse tree data. |
| l. | GUSER | – | user descriptions. |
| m. | GTDES | – | transaction descriptions. |
| n. | GLIST | – | list names and descriptions. |
| o. | GLABL | – | label names and descriptions. |
| p. | GOPEN | – | open files. |
| q. | GTADS | – | transformation aggregate descriptions. |
| r. | GTFDS | – | transformation field descriptions. |
| s. | GPFDS | – | field position descriptions. |
| t. | GTNAM | – | transformation field and aggregate names. |
| u. | GTFLD | – | transformation file descriptions. |

|  |  |  |  |
|---|---|---|---|
| v. | GTFLN | – | transformation file names. |
| w. | GDBASE | – | database names. |
| x. | GFEDES | – | error/message descriptions. |
| y. | GFEMES | – | error/message text. |
| z. | GQFIL | – | TAS batch query file names (refer to TAS Design Document.) |
| aa. | GUDES | – | TAS user descriptions (refer to TAS Design Documentation.) |

Each global data file is described in detail. All fields comprising a data structure are described. Where applicable (when the global data file is a C structure), an item description is provided illustrating the different fields defined for that global structure.

GADES (Aggregate Descriptions) – This structure contains descriptions of the data sets (or aggregates) of a specified file. Each item of the structure describes a data set and defines its relationships with other data sets. Item zero contains a description of the basic data set for the file. GADES is built by the DDL process and is used by most of the other processes. GADES is indexed by an aggregate ID (AGID) which is determined for a given name by an item-by-item search of the aggregate names (GANAM). GANAM and GTADS (transformation aggregate descriptions) are parallel to GADES and, therefore, are indexed by an AGID. Figure 54 is an item description.

| | | |
|---|---|---|
| Word 0 | GSIZE | |
| Word 1 | GPRNT | GATYP |
| Word 2 | GANUM | GAPTR |
| Word 3 | GLINK | GMPTR |
| Word 4 | GFNUM | |
| Word 5 | GFPTR | |
| Word 6 | GOCCS | |

Figure 54. GADES Item Description.

Field Descriptions

| | | |
|---|---|---|
| GSIZE | – | size of this aggregate's data, i.e., the number of characters of data for all single-valued visible fields in this data set. |
| GPRNT | – | pointer to the aggregate which is superordinate to this aggregate. |
| GATYP | – | Aggregate type:<br>0 = basic data set.<br>1 = repeating group.<br>2 = array. |
| GANUM | – | number of aggregates which are directly subordinate to this aggregate; zero if none. |
| GAPTR | – | pointer to the first aggregate in a linked list of aggregates which are directly subordinate to this aggregate; zero if none. |
| GLINK | – | pointer to the next aggregate in this aggregate's linked list of sibling aggregates. If GLINK is zero, this aggregate is the last in the list. |

| | | |
|---|---|---|
| GMPTR | – | pointer into an internal record map (GRMAP). GMPTR points into the data set node of the parent of this aggregate. GMPTR is relative to the start of the data set node. Refer to GRMAP for further explanation. |
| GFNUM | – | number of fields which are in this aggregate. |
| GFPTR | – | pointer to the first field in a sequential list of fields in this aggregate. GFPTR is an index to GFNAM and GFDES. |
| GOCCS | – | number of occurrences of an array. GOCCS is applicable for GATYP = 2 only. |

For the basic data set (index zero), the following fields are not applicable: GPRNT, GLINK, GMPTR, and GOCCS.

GANAM (Aggregate Names) – This array contains the names of repeating groups and/or arrays defined for a specified file. Each item contains an eight-character name. Item zero is not used. GANAM is built by the DDL process and is used by several other processes. GANAM is indexed by an aggregate ID (AGID) which can be determined for a given name by an item-by-item search. The aggregate descriptions (GADES) and transformation aggregate descriptions (GTADS) are parallel to GANAM and, therefore, are indexed by an AGID also. Figure 55 is a pictorial representation of GANAM.

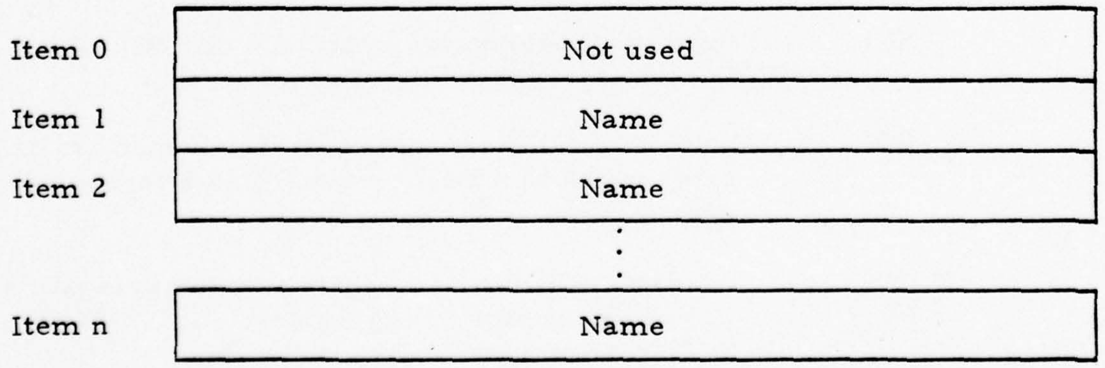


Figure 55. GANAM Pictorial Representation.

GFDES (Field Descriptions) – This structure contains descriptions of the fields in a specified file. Each item of the structure describes a field. Item zero is not used. GFDES is built by the DDL process and is used by most of the other processes. GFDES is indexed by a field ID (FID) which is determined for a given name by an item-by-item search of the field names (GFNAM). GFNAM is parallel to GFDES and, therefore, is indexed by an FID also. Figure 56 is an item description.

| | | |
|---|---|---|
| Word 0 | GSIZE | |
| Word 1 | GPRNT | GFTYP |
| Word 2 | GDTYP | GFATT |
| Word 3 | GIPTR | |

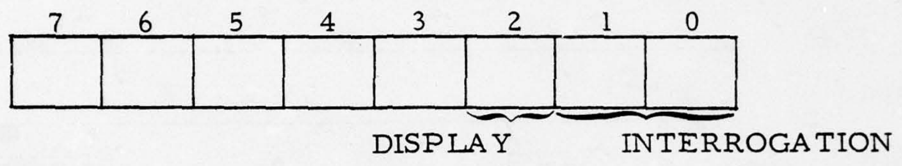
Figure 56. GFDES Item Description.

Field Descriptions

| | | |
|---|---|---|
| GSIZE | – | maximum size (number of characters) which a value of this field can be. Not applicable for GFTYP equal to two or three. |
| GPRNT | – | pointer to the aggregate under which this field is defined. GPRNT is an index to GANAM and GADES; i. e., an AGID. |
| GFTYP | – | Field Type:<br>0 = single-valued.<br>1 = multivalued.<br>2 = single-valued variable length.<br>3 = multivalued variable length. |
| GDTYP | – | data type of field's value(s):<br>0 = numeric.<br>1 = character.<br>2 = geographic. |

GFATT – field attributes. GFATT contains a bit string which defines the various attributes of the field. See figure 57.

GIPTR – pointer into an internal record map (GRMAP) or into an internal record data block (GRDAT). For GFTYP = 1 or 2, GIPTR points into the data set node of the field's parent aggregate. GIPTR is then relative to the start of the data set node. For GFTYP = 0, GIPTR is a pointer to GRDAT to this field's value relative to the start of the parent aggregate's data. Refer to GRMAP for further explanation.



INTERROGATION:

| Bits 1 | Bits 0 | | |
|---|---|---|---|
| 0 | 0 | = | not used. |
| 0 | 1 | = | KEY. |
| 1 | 0 | = | NKEY. |
| 1 | 1 | = | DEP. |

DISPLAY:

| Bit 2 | | |
|---|---|---|
| 0 | = | VIS. |
| 1 | = | INV. |

Figure 57. Field Attribute Bit Assignments.

GFNAM (Field Names) – This array contains the names of fields in a specified file. Each item contains an eight-character name. Item zero is not used. GFNAM is built by the DDL process and is used by several other processes. GFNAM is indexed by a field ID (FID) which can be determined for a given name by an item-by-item search. GFDES (field descriptions) is parallel to GFNAM and, therefore, is indexed by an FID also. Figure 58 is a pictorial representation of GFNAM.

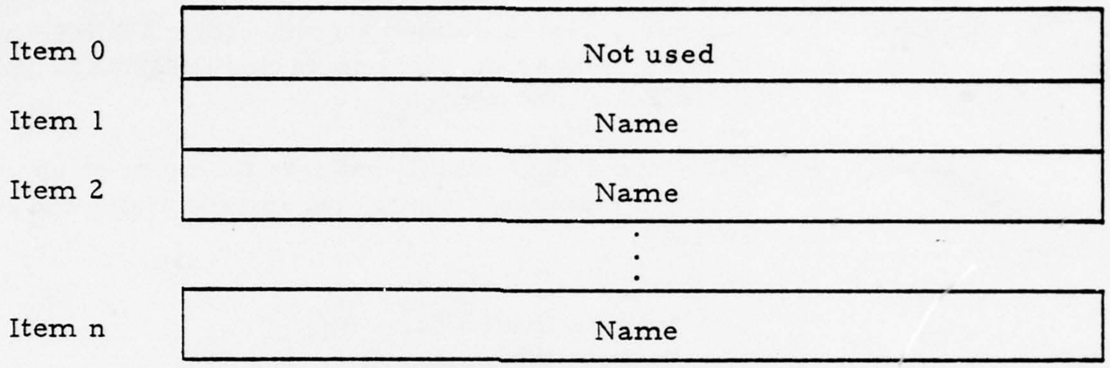


Figure 58. GFNAM Pictorial Representation.

<u>GFILD (File Descriptions)</u> – This structure contains descriptions of the files which can be accessed by ADAPT I. Each item of the structure describes a file. Item zero is not used. GFILD is built by the DDL process and is used by most of the other processes. GFILD is indexed by a file ID (FLID) which is determined for a given name by an item-by-item search of the file names (GFILN). GFILN is parallel to GFILD and, therefore, is indexed by an FLID also. Figure 59 is an item description.

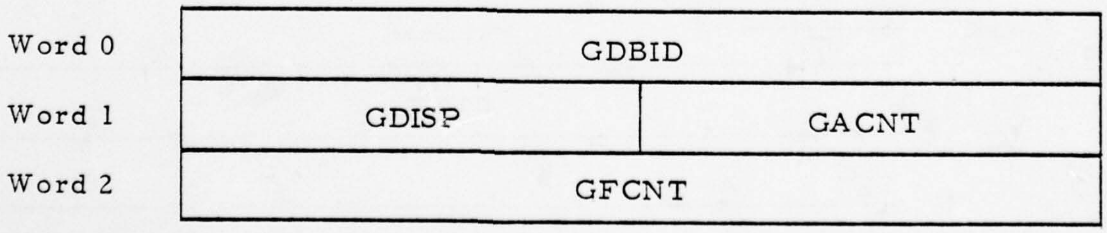


Figure 59. GFILD Item Description.

Field Descriptions

| | | |
|---|---|---|
| GDBID | – | data base ID (DBID) of the target system which actually contains the file being referenced. |
| GACNT | – | number of aggregates defined for this file. GACNT specifies the number of items contained in GANAM and GADES for this file. |
| GFCNT | – | number of fields defined for this file. GFCNT specifies the number of items contained in GFNAM and GFDES for this file. |
| GDISP | – | file disposition. GDISP reflects the mode of operation and location of the target system where the file resides. |

0 = local file.
1 = remote batch file.
2 = remote interactive file.

GFILN (File Names) – This array contains the names of the files which can be accessed by ADAPT I. Each item contains an eight-character name. Item zero contains in byte zero the actual number of files currently defined for ADAPT I. GFILN is built by the DDL process and is used by most other processes. GFILN is indexed by a file ID (FLID) which can be determined for a given name by an item-by-item search. GFILD (file descriptions) is parallel to GFILN and, therefore, is indexed by an FLID also. Figure 60 is a pictorial representation of GFILN.

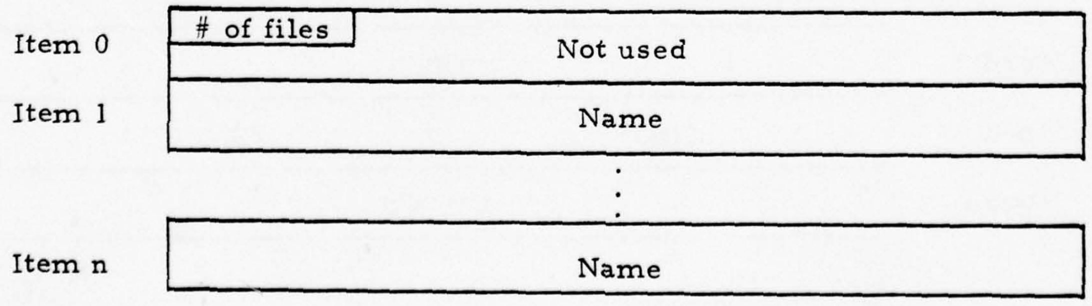


Figure 60. GFILN Pictorial Representation.

GRMAP (Internal Record Map) – This integer array is used to specify the locations of field value(s) for a specific record. Each data record which is sent by a target system to ADAPT is converted into an ADAPT internal record format. Each internal record consists of a record map (GRMAP) and one or more blocks of character data (GRDAT). GRMAP contains absolute pointers into GRDAT, as well as pointers into itself. GRMAP consists of two types of nodes: data set nodes and occurrence nodes. A data set node is related to a specific data set (or aggregate) and contains pointers to GRDAT and to occurrence nodes for subordinate aggregates. An occurrence node is also related to a specific aggregate and contains pointers to data set nodes. The data set node for the basic data set starts in word zero of GRMAP. GRMAP is built by TSOT and is used by the DISPLAY process. Descriptions of the two types of nodes in GRMAP follow.

Data Set Node – (See figure 61.)

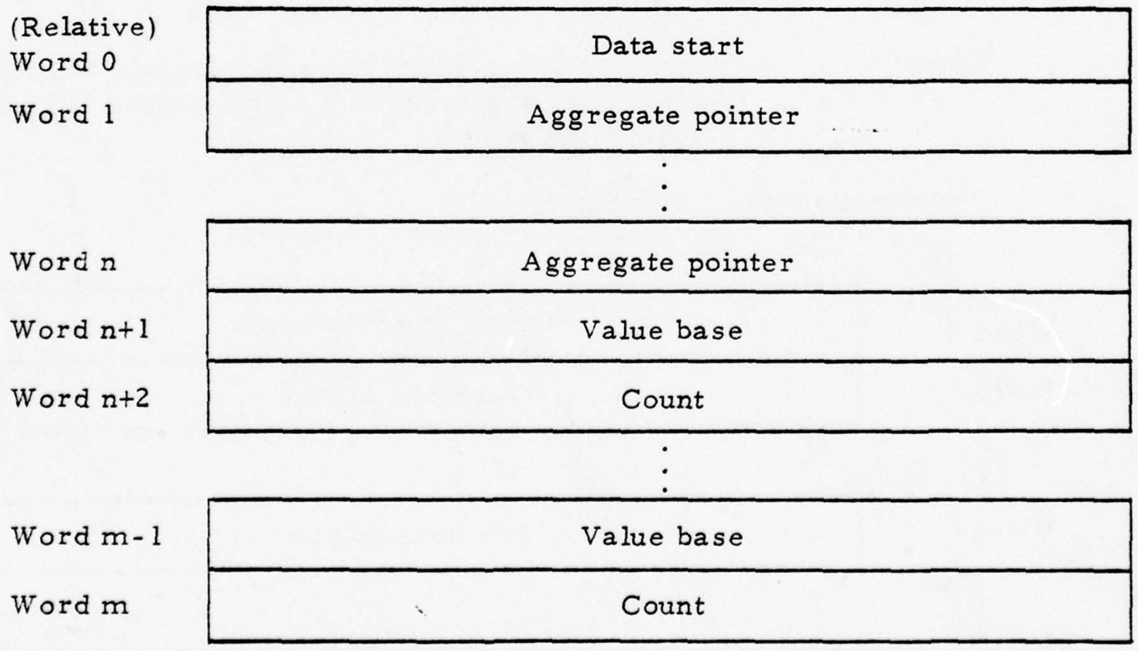


Figure 61. GRMAP Data Set Node Description.

Field Descriptions

| | | |
|---|---|---|
| Data start | – | pointer into GRDAT to the start of this aggregate's data values (for single-valued field only). If this aggregate has no single-valued fields or if there are no values for those fields, data start will be null. |
| Aggregate pointer | – | pointer into GRMAP to an occurrence node for a specific aggregate. There is an aggregate pointer for each aggregate which is directly subordinate to the aggregate whose data set node this is. This field is pointed to by field GMPTR of the aggregate descriptions (GADES). If the aggregate pointer is null, this aggregate has no occurrences at this point in the record. |
| Value base | – | pointer into GRDAT to the start of a field's data value(s). There is a value base/count for each multivalued and single-valued variable length field in the aggregate whose data set node this is. This field is pointed to by field GIPTR of the field descriptions (GFDES). If the field has no data, the value base is null. |
| Count | – | number of occurrences of a multivalued field or number of characters in a single-valued variable length field's data. |

Occurrence Node – (See figure 62.)

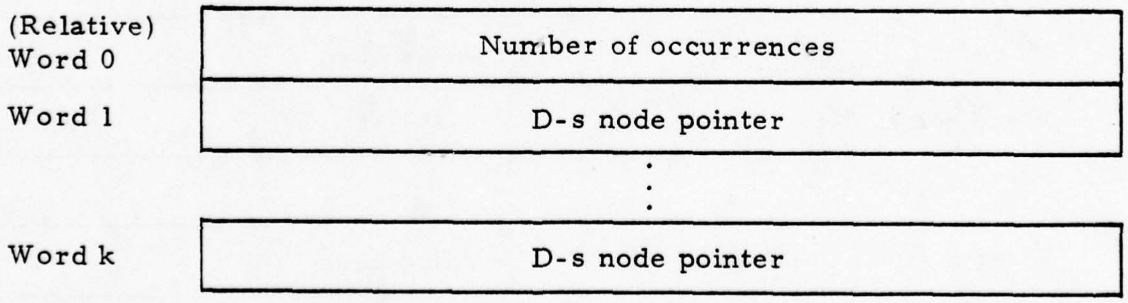


Figure 62. GRMAP Occurrence Node Description.

Field Descriptions

| | | |
|---|---|---|
| Number of occurrences | – | number of occurrences of the aggregate whose occurrence node this is. |
| D-s node pointer | – | pointer into GRMAP to a data set node for this occurrence of the aggregate. |

GMPHD (Internal Record Map Header) – This integer array contains information about the internal record map which it precedes when written onto a disk file. GMPHD contains size information which is required by ADAPT I when reading a sequence of record maps and record data. Figure 63 is a pictorial representation of GMPHD.

| | |
|---|---|
| Word 0 | File ID |
| Word 1 | Map size |
| Word 2 | Data start |
| Word 3 | Number blocks |
| Word 4 | Data size |

Figure 63. GMPHD Description.

Field Descriptions

| | | |
|---|---|---|
| File ID | – | pointer to the file to which the following record map (GRMAP) and its associated record data blocks (GRDAT) belong. The file ID (FLID) is an index to GFILN and GFILD. |
| Map size | – | number of words actually written out for the following record map. |
| Data start | – | word number within the record data disk file at which this record's data start. |

| | | |
|---|---|---|
| Number blocks | – | number of blocks of data actually written out for this record. Maximum block size is 512 bytes. |
| Data size | – | number of words actually written out for the final block of record data. |

GRDAT (Internal Record Data) – This character array is used to hold data values for a specific record. Index zero is not used. GRDAT is indexed by pointers stored in the record map (GRMAP) in combination with relative pointers stored in the field descriptions (GFDES). Single-valued fields which have no values will have ASCII blanks stored for them.

GTREE (Parse Tree) – This integer array contains the parse tree for a specific statement. Each UDL statement/command consists of terminal and nonterminal elements. In the parse tree, the pertinent elements in a statement correspond to terminal and nonterminal nodes. A nonterminal node contains a token type describing the nonterminal element and, perhaps, pointers to other nodes. A terminal node contains a token type describing the terminal element and a pointer into the parse tree data (GTDAT). A two-word header starts in word zero of GTREE. GTREE is built by the lexical analyzer portion of the EXEC, DDL, or TDL process. GTREE is used by most processes in ADAPT. Descriptions of the header and nodes follow.

Header – (see figure 64.)

| | |
|---|---|
| Word 0 | Initial pointer |
| Word 1 | Label flag |

Figure. 64. GTREE Header.

Field Descriptions

| | | |
|---|---|---|
| Initial pointer | – | pointer into GTREE to the primary nonterminal node for this statement. |
| Label flag | – | flag to specify whether this statement has a label. 0 = no; 1 = yes. If yes, the label is stored in the first eight bytes of the parse tree data (GTDAT). |

Nonterminal Node (Type 1) – (See figure 65.)

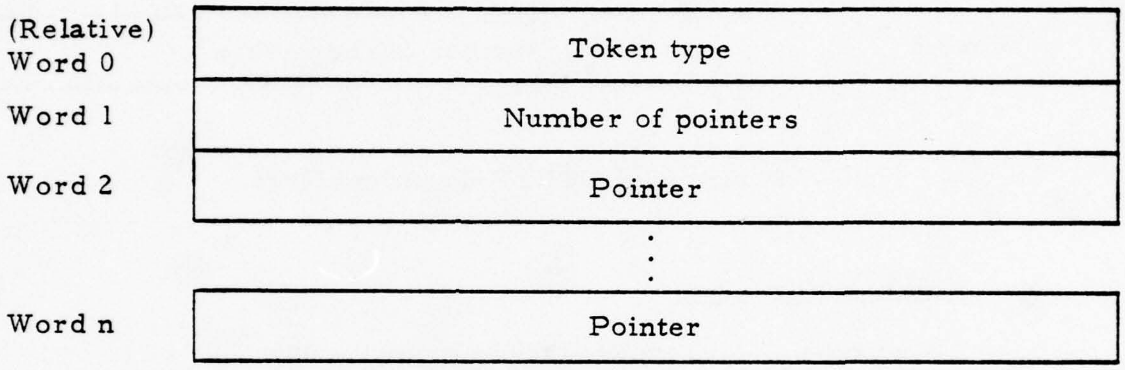


Figure 65. GTREE Nonterminal Node (Type 1).

Nonterminal Node (Type 2) – (See figure 66.)

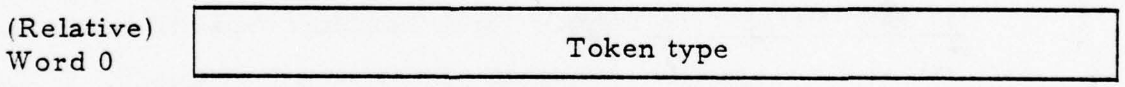


Figure 66. GTREE Nonterminal Node (Type 2).

Field Descriptions

| | | |
|---|---|---|
| Token type | – | nonterminal token type (100 – 599). For keywords with no arguments (e.g., ECHO), a type 2 node is stored in the parse tree. |

| | | |
|---|---|---|
| Number of pointers | – | number of pointers following this word (can be zero). |
| Pointer | – | pointer into GTREE to another node. |

Terminal Node – (See figure 67.)

| (Relative) | |
|---|---|
| Word 0 | Token type |
| Word 1 | Data pointer |
| Word 2 | Number of characters |

Figure 67. GTREE Terminal Node.

Field Descriptions

| | | |
|---|---|---|
| Token type | – | terminal token type (1 – 99). |
| Data pointer | – | pointer into the parse tree data (GTDAT) to the data associated with this parse tree node. |
| Number of characters | – | number of characters of data stored in GTDAT for this parse tree node. |

GTDAT (Parse Tree Data) – This character array is used to hold data values for a specific statement. Index zero is not used. GTDAT is indexed by pointers stored in the parse tree (GTREE). GTDAT is built by the lexical analyzer portion of either the EXEC, TDL or DDL process. GTDAT is used by most processes in ADAPT I.

GUSER (User Descriptions) – This structure contains information pertinent to all users that utilize ADAPT I. The establishment of an ADAPT I user is accomplished by the ADAPT I ADUSER Program. During this process, the user name is entered. A user must be established in GUSER prior to using ADAPT I. Internally, users are recognized by an index (UID) into this structure. A UID of zero (item 0 of GUSER) implies the ADAPT I superuser. Figure 68 is an item description.

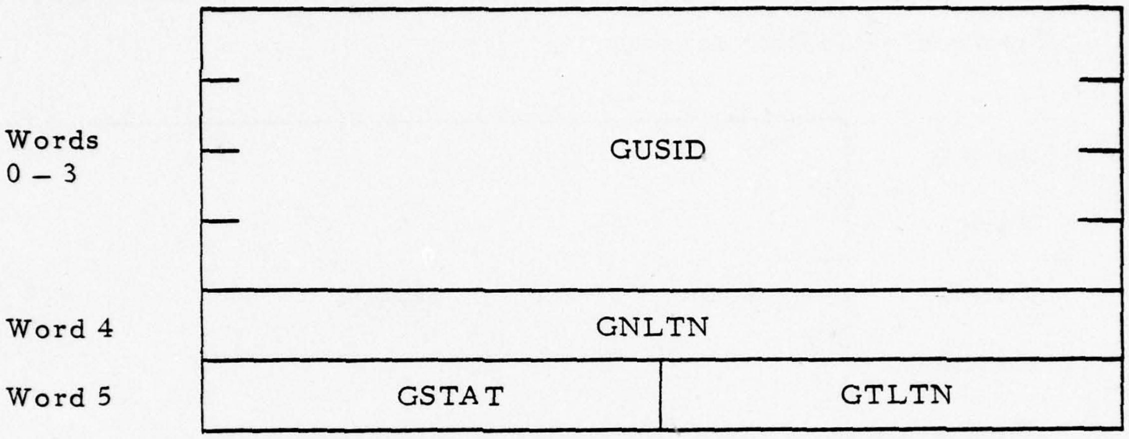


Figure 68. GUSER Item Description.

<u>Field Descriptions</u>

| | | |
|---|---|---|
| GUSID | – | user identifier, 8 characters. Must be unique within GUSER. |
| GSTAT | – | contains user status: 0 = user is not logged onto ADAPT I; 1 = user is logged onto ADAPT I. |
| GNLTN | – | contains the next logical transaction number (LTN) for this user. LTNs are applicable for batch transactions only. A batch transaction is uniquely identified for a user by the concatenation of the user's UID with the appropriate LTN. |
| GTLTN | – | contains the total number of current transactions. |

<u>GTDES (Transaction Descriptions)</u> – This structure contains information pertinent to ADAPT I logical transactions. Logical transactions are applicable for batch queries only. Entries into GTDES are

made by the Target System Input Generator: Batch (TSIGB) process and are removed by the VIEW or DELETE processes. Entries in GTDES are identifiable by a ten-character TAS JOBID transaction number. GTDES entries remain in GTDES until a target system has responded and a user has perused (via the VIEW command) the associated batch output. Figure 69 is an item description.

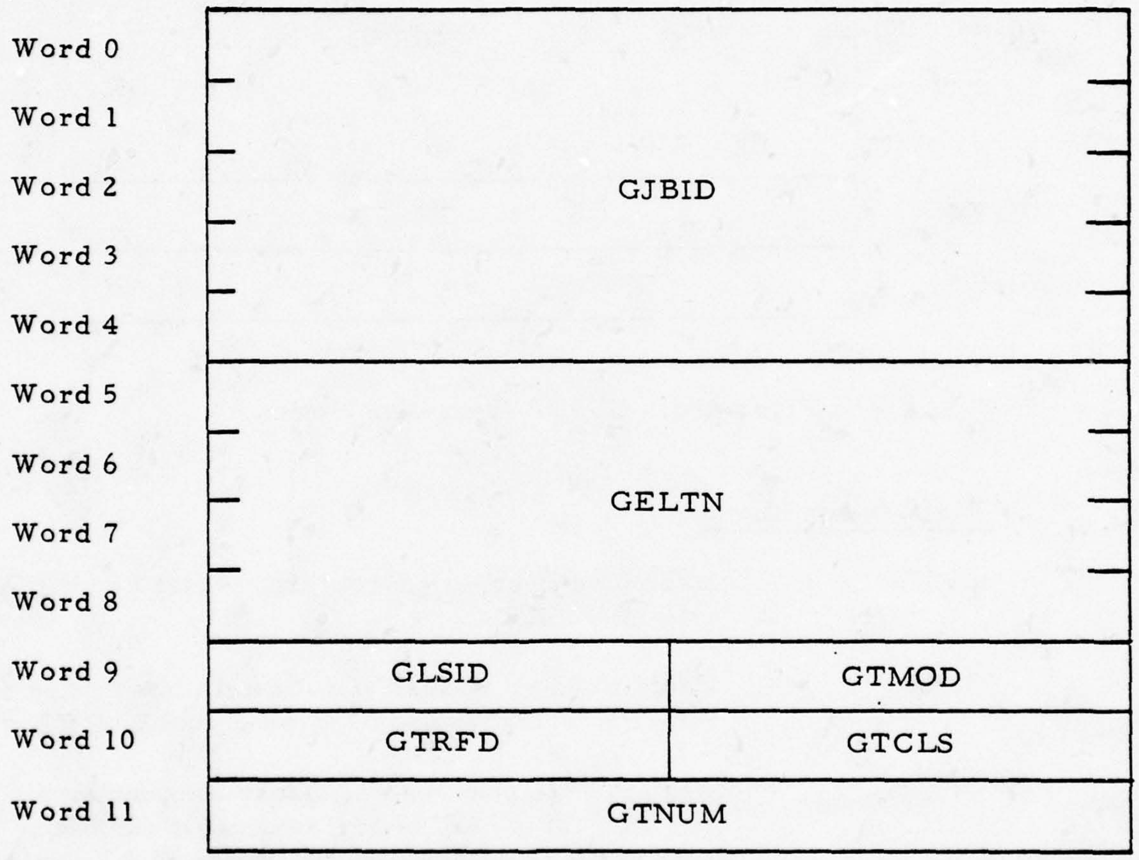


Figure 69. GTDES Item Description.

Field Descriptions

| | | |
|---|---|---|
| GJBID | – | ten character, TAS assigned job identifier for the transaction. |

| | | |
|---|---|---|
| GELTN | – | external logical transaction number, eight characters and unique in GTDES. User identifier, LTN, and a possible subtransaction number are contained in this field. |
| GLSID | – | record list index. Points to the appropriate GLIST item which will represent (either as an explicit or implicit list) the transaction output. |
| GTMOD | – | transaction mode: |
| | | 0 = slot is empty. |
| | | 1 = active transaction, waiting for the target system response. |
| | | 2 = answered transaction. |
| | | 3 = aborted transaction. |
| | | 4 = logged out or lost transaction |
| | | 5 = active multi-interrogation instance |
| | | 6 = answered multi-interrogation instance |
| | | 7 = aborted multi-interrogation instance |
| | | 8 = logged out or lost multi-interrogation instance |
| GTRFD | – | target system file identifier. |
| GTCLS | – | response classification (if answered). |
| GTNUM | – | number of interrogations for this transformation. |

Item 0 of GTDES specifies the number of logical transaction entries currently residing in GTDES.

GLIST (List Names and Descriptions) – This structure contains information pertinent to UDL record lists. A list can originate either as an explicit list, due to a user entering a SAVE command, or as an implicit list generated internally by ADAPT I, caused by a user entering a DISPLAY statement. The explicit list will exist indefinitely until the user deletes it. An implicit list exists only for the duration of time required to display its records. All list names are unique for a given user. Internal list names will be generated uniquely in order to avoid conflict with user specified names. Entries into GLIST are made by the Target System Input Generator processes (TSIGI and TSIGB). Figure 70 is an item description.

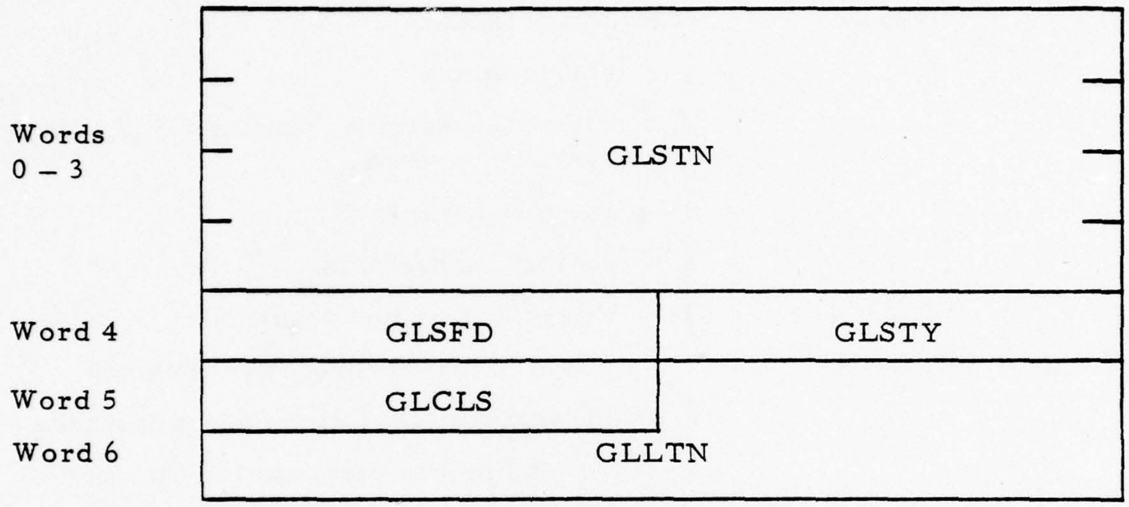


Figure 70. GLIST Item Description.

<u>Field Descriptions</u>

| | | |
|---|---|---|
| GLSTN | – | list name, 1 – 8 characters. Must be unique within GLIST. |
| GLSFD | – | contains file identifier (FLID, refer to global data structures GFILN and GFLID) of file from which list records originated. |
| GLSTY | – | list type: 0 = implicit list; 1 = explicit list; 2 = completed explicit list; 3 = delivered explicit list. |
| GLCLS | – | response classification. |
| GLLTN | – | logical transaction number (ASCII) assigned to the query by ADAPT. |

Item 0 of GLIST is used to contain the number of lists currently residing in GLIST.

GLABL (Label Names and Descriptions) – This structure contains information pertinent to user statement labels that are defined during any given logon session. Labels are applicable on FIND statements only (for ADAPT I) and exist only for the duration of the user's logon session. That is, once a user quits ADAPT I, all FIND statement labels are deleted. Also a user may only have a specified number of labels active at any one time, the oldest label being deleted in order to accommodate the new label. Labels have significance for dependent FIND statement references and for DISPLAY and SAVE record source specifications. Labels are stored by the Target System Input Generator (TSIGI and TSIGB) processes and are all deleted when the user initiates an ADAPT I QUIT command. Figure 71 is an item description.

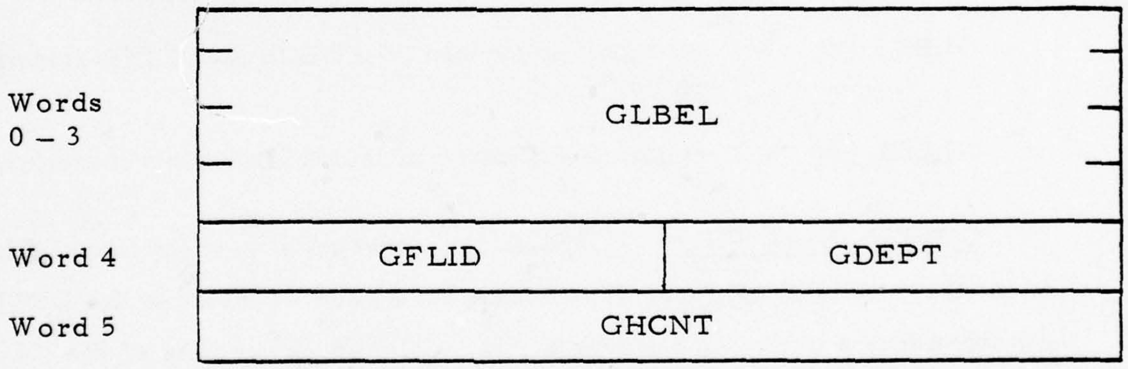


Figure 71. GLABL Item Description.

Field Descriptions

GLBEL – label, 1 – 8 characters. Must be unique within GLABL.

| | | |
|---|---|---|
| GFLID | – | contains the file identifier (FLID, refer to global data structures GFILN and GFILD) of the file referenced by the FIND statement (either explicitly via IN filename or implicitly via SOURCE (tag)). |
| GDEPT | – | indicator for FIND statement type: 0 = label is associated with an independent FIND statement; ≠ 0 = label is associated with a dependent FIND statement. For this case, GDEPT contains an index into GLABL to the referenced FIND statement. |
| GHCNT | – | hit count for the FIND statement for interactive file references. |

Item 0 of GLABL (field GFLID) is used to contain the number of labels currently residing in GLABL as well as the current state of the user's interactive host connection. The following GLABL fields are used as follows:

| | | |
|---|---|---|
| GLBEL [0] | – | contains the current interactive host ID. |
| GLBEL [1] | – | contains index into GLABL to last FIND statement entry. |
| GLBEL [2] | – | contains indicator of last SOLIS file referenced. |

<u>GOPEN (Open Files)</u> – This array contains a current list of file identifiers (FLID) of those files which have been opened via the OPEN command by a user. Whenever a file is closed, by means of the CLOSE command, the file index is removed from GOPEN. The file identifiers can be used to index files GFILN, GFILD, GTFLD, and GTFLN. Access authorization to the target system and the particular file for both the user and the terminal must be established before a file can be opened. The GOPEN file is maintained by the OPENCLOSE process. The GOPEN file is like the GLABL file in that it only exists for the duration of a user's logon session.

GTADS (Transformation Aggregate Descriptions) – This structure contains the information necessary to obtain the transformation aggregate names from the array GTNAM, which contains both field and aggregate transformation names. The data sets (or aggregates) of a specified file normally have names by which the target system (where the file resides) knows the aggregates. These names are stored in array GTNAM. Pointers and character counters stored in GTADS allow easy retrieval of the aggregate names from GTNAM. Each item of the structure is indexed by an aggregate ID (AGID) which is usually determined for a given name by an item-by-item search of the aggregate names (GANAM). GTADS is parallel to GANAM and GADES, which are also indexed by AGID. Item zero is not used. Figure 72 is an item description of GTADS

| | | |
|---|---|---|
| Word 0 | GPTR1 | |
| Word 1 | GNCH1 | Not used |

Figure 72. GTADS Item Description.

Field Descriptions

| | | |
|---|---|---|
| GPTR1 | – | absolute pointer to the aggregate transformation name stored in GTNAM. |
| GNCH1 | – | number of characters in the aggregate transformation name. |

GTFDS (Transformation Field Descriptions) – This structure consists of pointers and character counters which are the mechanism for retrieving transformation field names stored in transformation name array GTNAM. Each field in a specified file is primarily described for ADAPT I purpose by four parallel files which are:

a. GFDES – field descriptions.

b. GFNAM – field names.

c. GPFDS – position field descriptions.

d. GTFDS – transformation field descriptions.

These four files are indexed by a field ID (FID) which is usually determined for a given field name by an item-by-item search of the field names (GFNAM). GTFDS is built by the TDL process and is used by those processes which are involved with the transformations of local queries to target host systems. Figure 73 is an item description of GTFDS.

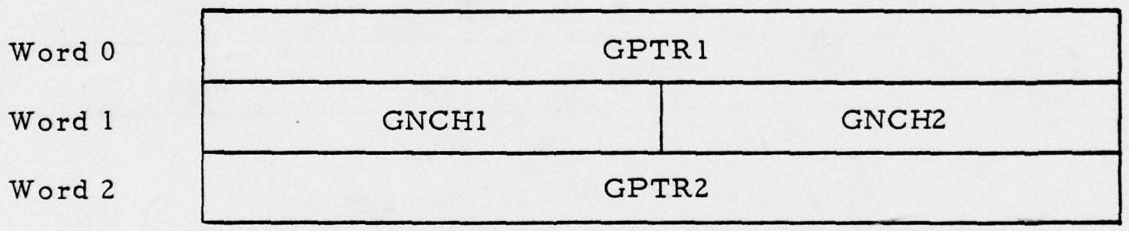


Figure 73. GTFDS Item Description.

Field Descriptions

| | | |
|---|---|---|
| GPTR1 | – | absolute pointer to the first character of the first transformation field name which is stored in the transformation name file, GTNAM. |
| GNCH1 | – | number of characters in the first transformation field name. |

| | |
|---|---|
| GPTR2 – | absolute pointer to the first character of the second transformation field name which is stored in the transformation name file, GTNAM. |
| GNCH2 – | number of characters in the second transformation field name. |

<u>GPFDS (Field Position Descriptions)</u> – This structure contains information concerning field position descriptions for files which are position dependent. Each item of the structure describes a field and is indexed by a field identifier (FID). Four tables are parallel in the physical order of storing field information: GFNAM, GFILD, GTFDS, and GPFDS. GPFDS is built by the TDL process and is used by processes involved in transformation of queries and responses. The file GPFDS is only built when a file is position dependent; in other words, the file GPFDS is necessary when responses to queries on the file data are constructed in a position dependent format. Figure 74 is an item description of GPFDS

| | | |
|---|---|---|
| Word 0 | GFPOS | |
| Word 1 | GNUMO | GPRNM |

Figure 74. GPFDS Item Description.

<u>Field Descriptions</u>

| | |
|---|---|
| GFPOS – | field character position. The character location within a response record where the field information appears. |
| GNUMO – | number of occurrences for fields which are multivalued. |
| GPRNM – | response number in which this field's data will appear if there are multiple print requests and therefore multiple responses. |

<u>GTNAM (Transformation Field and Aggregate Names)</u> – This array consists of the transformation names for aggregates and fields for a specific file. The names which are stored in GTNAM are referenced by pointers and counters stored in the transformation aggregate descriptions file (GTADS) and transformation field descriptions file (GTFDS). GTNAM is built by the TDL process and is used by those processes involved in transformations.

<u>GTFLD (Transformation File Descriptions)</u> – This structure contains transformation information pertinent to the files which can be accessed by ADAPT I. Each item of the structure describes a file and is indexed by a file identifier (FLID) which corresponds to the physical order of GFILN, GFILD and GTFLN. Item zero is not used. GTFLD is maintained by the TDL process and used by those processes involved in the transformations of queries and responses. Figure 75 is an item description of GTFLD.

| | |
|---|---|
| Word 0 | GDBID |
| Word 1 | GTOTC |
| Word 2 | GKEYF |

Figure 75. GTFLD Item Description.

<u>Field Descriptions</u>

| | | |
|---|---|---|
| GDBID | – | database identifier of the target system where the file resides. |
| GTOTC | – | exact number of characters per record in a character position dependent file. Value is zero if the file is not position dependent. |
| GKEYF | – | field identifier (FID) of key field, or zero if none. |

GTFLN (Transformation File Names) – This two-dimensional array contains the transformation filenames for the files which can be accessed by ADAPT I. The transformation filename is the name by which the file is recognized and acknowledged by the target host system with which it is associated. Each item contains a ten character name except for item zero which is not used. GTFLN is maintained by the TDL process and is used by those processes involved in transformations. GTFLN as well as parallel structured files GFILN, GTFLD, and GFILD is indexed by a file ID (FLID) which can be determined for a given file name by an item-by-item search of GFILN. Figure 76 is a pictorial representation of GTFLN.

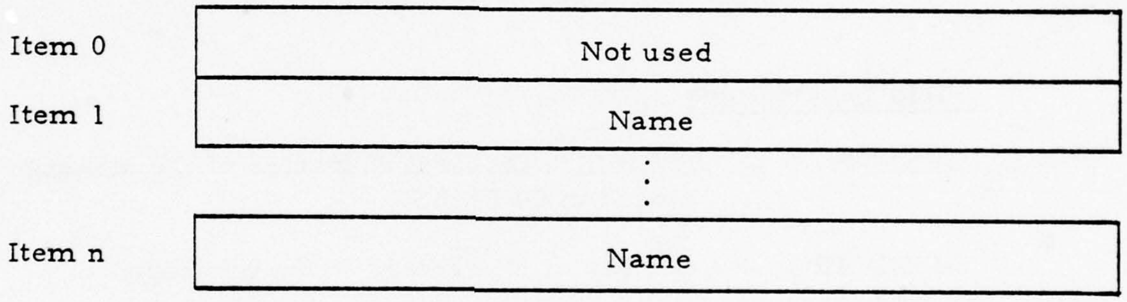


Figure 76. GTFLN Pictorial Representation.

GDBASE (Data Base Names) – This two-dimensional character array contains the names of the database names which ADAPT I can access. The use of this array prevents the storage and parameter passage of six character names and allows the use of a host ID index. The actual C language is given below:

```
char *GDBASE {" ", "RYETIP", "DIAOLS", "SOLIS",
"ISSPIC", 0};
```

GFEDES (Error/Message Descriptions) – GFEDES is a structure containing pertinent information concerning messages stored in character message buffer GFEMES. This file is built by ERRINIT and used in conjunction with GFEMES by global function GFERROR to output instructional or diagnostic messages to a user's terminal. Figure 77 is an item description of GFEDES.

| | | |
|---|---|---|
| Word 0 | GFEPTR | |
| Word 1 | GFECHAR | GFNAME |

Figure 77. GFEDES Item Description.

Field Descriptions

| | | |
|---|---|---|
| GFEPTR | – | Pointer to the first character of the message stored in GFEMES. |
| GFECHAR | – | number of characters in the message. |
| GFNAME | – | relative pointer to the start of the inserted data if such information is pertinent; otherwise, minus one. |

GFEMES (Error/Message Text) – GFEMES is a character array containing the actual ADAPT messages which, when used in conjunction with GFEDES, is the mechanism by which GFERROR outputs instructional and diagnostic messages to a user's terminal. The ERRINIT process builds GFEMES using global file ERRORCODES as input.

# APPENDIX A

## ADAPT I YACC SPECIFICATIONS

YACC is utilized for front-end development in ADAPT I. This appendix presents the UDL/DDL/TDL grammars as YACC specifications. YACC accepts left associative LR(1) grammars as input and resolves any syntactic ambiguities if they occur. The output from YACC is a parser program and an LR(1) parse table representing the grammar. Not shown in the YACC specifications presented herein are the user supplied lexical analyzer and parse tree generators.

ADAPT I utilizes YACC for three separate front-ends: 1) a front-end for parsing UDL statements and commands, 2) a front-end for DDL statements, and 3) a front-end for TDL statements.

## UDL YACC Specification

```
%token  ';' 0
%token  NAME 1
%token  CHARCONS 30     NUMCONS 31      GEOCONS 32      INTEGER 33
%token  OR 100  AND 101 XOR 102 NOR 103 NOT 104
%token  EQ 140  GT 141  LT 142  GE 143  LE 144  WRG 145 ORG 146
%token  INSIDE 150      OUTSIDE 151     ALONG 152
%token  IN 300  SOURCE 301      CIRCLE 302      POLY 303        ROUTE 304
%token  REMOTE 305      TREE 307        LIST 308        ECHO 309
%token  VALIDATE 310    ACTION 311      SNAME 312       GEOLIST 313
%token  DISPLIST 314    FNAMLIST 315
%token  SCOPE 316       FILE 317        LABEL 318       TRANS   320     KEY 406
%token  LOCAL 412       BATCH 413       INTER 414       POSITION 420
%token  DATABASE 422
%token  FIND 500        DISPLAY 501
%token  QUIT 520        OPEN 521        CLOSE 522       EXECUTE 523
%token  MODE 524        SAVE 525        DELETE 526      VIEW 527
%token  SCHEMA 550      TRANFILE 560
%token  '(' 490 ')' 491 ',' 492
%token  LEXERR 600
%%                                      /* beginning of rules section */
list:                                   /* list is the start symbol */
                                        /* empty */
        | list stat ';'                 /* statement */
                = { psfinsh($2) ;}
        | list error ';'                /* syntax error */
                = { psfinsh(0) ; }  ;
stat:
        FIND sclause selcrit            /* FIND statement */
                = { $$ = pstree2(FIND,$2,$3) ;}
        | DISPLAY SOURCE '(' NAME ')'   /* DISPLAY statement */
                = { $$ = pstree1(DISPLAY,$4) ;}
        | DISPLAY SOURCE '(' NAME ')' dlist
                = { $$ = pstree2(DISPLAY,$4,$6) ;}
        | QUIT                  /* QUIT command */
                = { $$ = pstreeno(QUIT) ;}
        | OPEN fnlist                   /* OPEN command */
                = { $$ = pstree1(OPEN,$2) ;}
        | SAVE SOURCE '(' NAME ')' NAME /* SAVE command */
                = { $$ = pstree2(SAVE,$4,$6) ;}
        | DELETE dclause                /* DELETE command */
                = { $$ = pstree1(DELETE,$2) ;}
        | VIEW vclause                  /* VIEW command */
                = { $$ = pstree1(VIEW,$2) ;}
        | CLOSE fnlist                  /* CLOSE command */
                = { $$ = pstree1(CLOSE,$2) ;}
        | CLOSE
                = { $$ = pstree0(CLOSE) ;}
        | MODE modespec                 /* MODE command */
                = { $$ = pstree1(MODE,$2) ;}
        | EXECUTE namespec              /* EXECUTE command */
                = { $$ = pstree1(EXECUTE,$2) ;}
        | EXECUTE namespec echospec
                = { $$ = pstree2(EXECUTE,$2,$3)  ;}
        | TRANFILE namclause dbclause   /* TRANFILE statement */
                = { $$ = pstree2(TRANFILE,$2,$3) ;}
        | TRANFILE namclause dbclause posclause
                = { $$ = pstree3(TRANFILE,$2,$3,$4) ;}
        | TRANFILE namclause dbclause keyclause
                = { $$ = pstree3(TRANFILE,$2,$3,$4) ;}
        | TRANFILE namclause dbclause posclause keyclause
                = { $$ = pstree4(TRANFILE,$2,$3,$4,$5) ;}
        | SCHEMA NAME disparg           /* SCHEMA statement */
                = { $$ = pstree2(SCHEMA,$2,$3) ;} ;
sclause:                                /* source clause for FIND statement */
        IN NAME
                = { $$ = pstree1(IN,$2) ;}
        | SOURCE '(' NAME ')'
                = { $$ = pstree1(SOURCE,$3) ;} ;
selcrit:                                /* selection criteria */
        selclause
        | selcrit OR selclause
                = { $$ = pstree2(OR,$1,$3) ;} ;
selclause:
        selphrase
        | selclause AND selphrase
                = { $$ = pstree2(AND,$1,$3) ;} ;
selphrase:
        selfactor
        | selphrase XOR selfactor
                = { $$ = pstree2(XOR,$1,$3) ;}
        | selphrase NOR selfactor
                = { $$ = pstree2(NOR,$1,$3) ;} ;
selfactor:
        selprimary
        | NOT selprimary
                = { $$ = pstree1(NOT,$2) ;} ;
selprimary:
        selterm
        | '(' selcrit ')'
                = { $$ = $2 ;}
        | NAME '(' selcrit ')'
                = { $$ = pstree2(SCOPE,$1,$3) ;} ;
selterm:
        fterm EQ datacons
                = { $$ = pstree2(EQ,$1,$3) ;}
        | fterm GT datacons
                = { $$ = pstree2(GT,$1,$3) ;}
```

```
         | fterm LT datacons
                = { $$ = pstree2(LT,$1,$3) ;}
         | fterm GE datacons
                = { $$ = pstree2(GE,$1,$3) ;}
         | fterm LE datacons
                = { $$ = pstree2(LE,$1,$3) ;}
         | fterm WRG datacons ',' datacons
                = { $$ = pstree3(WRG,$1,$3,$5) ;}
         | fterm ORG datacons ',' datacons
                = { $$ = pstree3(ORG,$1,$3,$5) ;}
         | fterm INSIDE geoexp
                = { $$ = pstree2(INSIDE,$1,$3) ;}
         | fterm OUTSIDE geoexp
                = { $$ = pstree2(OUTSIDE,$1,$3) ;}
         | fterm ALONG geoexp
                = { $$ = pstree2(ALONG,$1,$3) ;} ;
fterm:
        NAME
        | subname ;
subname:                                        /* subscripted field name */
        NAME '(' INTEGER ')'
                = { $$ = pstree2(SNAME,$1,$3) ;}
        | NAME '(' INTEGER ',' INTEGER ')'
                = { $$ = pstree3(SNAME,$1,$3,$5) ;}
        | NAME '(' INTEGER ',' INTEGER ',' INTEGER ')'
                = { $$ = pstree4(SNAME,$1,$3,$5,$7) ;} ;
datacons:                                       /* data constant */
        NUMCONS
        | INTEGER
        | CHARCONS
        | GEOCONS ;
geoexp:                                         /* geographic expression */
        CIRCLE '(' datacons ',' GEOCONS ')'
                = { $$ = pstree2(CIRCLE,$3,$5) ;}
        | POLY '(' gelist ')'
                = { $$ = pstree1(POLY,$3) ;}
        | ROUTE '(' datacons ',' gelist ')'
                = { $$ = pstree2(ROUTE,$3,$5) ;} ;
gelist:                                         /* geographic constant list */
        glist
                = { $$ = pslstdon(GEOLIST) ;} ;
glist:
        GEOCONS
                = { pslstnew($1) ;}
        | glist ',' GEOCONS
                = { pslstnxt($3) ;} ;
dlist:                                          /* display list */
        dislist
                = { $$ = pslstdon(DISPLIST) ;} ;
```

```
dislist:
        dispelem
                = { pslstnew($1) ;}
        | dislist ',' dispelem
                = { pslstnxt($3) ;} ;
dispelem:                                  /* display element */
        CHARCONS
        | fterm
        | TREE '(' NAME ')'
                = { $$ = pstree1(TREE,$3) ;} ;
dclause:                                   /* DELETE clause */
        LIST '(' NAME ')'
                = { $$ = pstree1(LIST,$3) ;}
        | TRANFILE '(' NAME ')'
                = { $$ = pstree1(TRANFILE,$3) ; }
        | SCHEMA '(' NAME ')'
                = { $$ = pstree1(SCHEMA,$3) ;} ;
vclause:                                   /* VIEW clause */
        LIST
                = { $$ = pstreeno(LIST) ;}
        | FILE
                = { $$ = pstreeno(FILE) ; }
        | LABEL
                = { $$ = pstreeno(LABEL) ; }
        | TRANS
                = { $$ = pstree0(TRANS) ; }
        | TRANS '(' INTEGER ')'
                = { $$ = pstree1(TRANS,$3) ; }
        | TRANFILE '(' NAME ')'
                = { $$ = pstree1(TRANFILE,$3) ; }
        | SCHEMA '(' NAME ')'
                = { $$ = pstree1(SCHEMA,$3) ; } ;
fnlist:                                    /* file name list */
        filelist
                = { $$ = pslstdon(FNAMLIST) ;} ;
filelist:
        NAME
                = { pslstnew($1) ;}
        | filelist ',' NAME
                = { pslstnxt($3) ;} ;
modespec:                                  /* MODE specification */
        VALIDATE
                = { $$ = pstreeno(VALIDATE) ;}
        | ACTION
                = { $$ = pstreeno(ACTION) ;} ;
namespec:                                  /* unix-file specification for EXECUTE */
        CHARCONS
        | SCHEMA'(' CHARCONS ')'
```

```
                        = { $$ = pstree1(SCHEMA,$3) ;}
                | TRANFILE '(' CHARCONS ')'
                        = { $$ = pstree1(TRANFILE,$3) ; } ;
echospec:
                ECHO
                        = { $$ = pstreeno(ECHO) ;} ;
disparg:
                LOCAL
                        = { $$ = pstreeno(LOCAL) ; }
                | REMOTE '(' BATCH ')'
                        = { $$ = pstreeno(BATCH) ; }
                | REMOTE '(' INTER ')'
                        = { $$ = pstreeno(INTER) ; } ;
namclause:
                NAME
                | nmclause ;
nmclause:
                NAME '(' CHARCONS ')'
                        = { $$ = pstree2(SNAME,$1,$3) ; } ;
dbclause:
                DATABASE '(' NAME ')'
                        = { $$ = pstree1(DATABASE,$3) ; } ;
posclause:
                POSITION '(' INTEGER ')'
                        = { $$ = pstree1(POSITION,$3) ; } ;
keyclause:
                KEY '(' NAME ')'
                        = { $$ = pstree1(KEY,$3) ; } ;
%%                                              /* beginning of programs */
```

## DDL YACC Specification

```
%token  ';'0
%token  NAME 1
%token  INTEGER'.33 REMOTE 305
%token  ATT 400 SV 401 MV 402 CHAR 403 NUM 404 GEO 405 KEY 406 NKEY 407
%token  DEP 408 VIS 409 INV 410 V 411 LOCAL 412
%token  SCHEMA 550 ESCHEMA 551 FIELD 552 ARRAY 553
%token  EARRAY 554 RGROUP 555 ERGROUP 556
%token  '(' 490 ')' 491 ',' 492
%token  LEXERR 600
%%                                      /* beginning of rules section */
list:                                   /* list is the start symbol */
                                        /* empty */
        | list stat ';'                 /* statement */
                = { psfinish($2) ;}
        | list error ';'                /* syntax error */
                = { psfinish(0) ;} ;
stat:
        SCHEMA NAME disparg             /* SCHEMA statement */
                = { $$ = pstree2(SCHEMA,$2,$3) ;}
        | ESCHEMA                       /* ESCHEMA statement */
                = { $$ = pstreeo(ESCHEMA) ;}
        | FIELD NAME ftyp dtyp ATT '(' iatt ',' datt ')' size /*FIELD statement*/
                = { $$ = pstree6(FIELD,$2,$3,$4,$7,$9,$11) ;}
        | ARRAY NAME INTEGER            /* ARRAY statement */
                = { $$ = pstree2(ARRAY,$2,$3) ;}
        | ARRAY NAME INTEGER NAME
                = { $$ = pstree3(ARRAY,$2,$3,$4) ;}
        | EARRAY                        /* EARRAY statement */
                = { $$ = pstreeno(EARRAY) ;}
        | RGROUP NAME                   /* RGROUP statement */
                = { $$ = pstree1(RGROUP,$2) ;}
        | RGROUP NAME NAME
                = { $$ = pstree2(RGROUP,$2,$3) ;}
        | ERGROUP                       /* ERGROUP statement */
                = { $$ = pstreeno(ERGROUP) ;} ;
ftyp:                                   /* field type */
        SV
                = { $$ = pstreeno(SV) ;}
        | MV
                = { $$ = pstreeno(MV) ;} ;
dtyp:                                   /* data type */
        CHAR
                = { $$ = pstreeno(CHAR) ;}
        | NUM
                = { $$ = pstreeno(NUM) ;}
        | GEO
                = { $$ = pstreeno(GEO) ;} ;
iatt:                                        /* interrogation attribute */
        KEY
                = { $$ = pstreeno(KEY) ;}
        | NKEY
                = { $$ = pstreeno(NKEY) ;}
        | DEP
                = { $$ = pstreeno(DEP) ;} ;
datt:                                        /* display attribute */
        VIS
                = { $$ = pstreeno(VIS) ;}
        | INV
                = { $$ = pstreeno(INV) ;} ;
size:                                        /* field size */
        INTEGER
        | INV
                = { $$ = pstreeno(V) ;} ;
disparg:
        LOCAL
                = { $$ = pstreeno(LOCAL) ; }
        | REMOTE '(' BATCH ')'
                = { $$ = pstreeno(BATCH) ; }
        | REMOTE '(' INTER ')'
                = { $$ = pstreeno(INTER) ; } ;
```

## TDL YACC Specification

```
%token  ';' 0
%token  NAME 1
%token  CHARCONS 30     INTEGER 33
%token  SNAME 312
%token  KEY 406 DATABASE 422    POSITION 420    POS 421
%token  FIELD 552       ARRAY 553       RGROUP 555      TRANFILE 560
%token  ETRAN 561
%token  '(' 490 ')' 491 ',' 492
%token  LEXERR 600
%%                                      /* beginning of rules section */
list:                                   /* list is the start symbol */
                                        /* empty */
        | list stat ';'       /* statement */
                = { psfinsh($2);}
        | list error ';'                /* syntax error */
                = { psfinsh(0) ; } ;
stat:
        TRANFILE namclause dbclause
                = { $$ = pstree2(TRANFILE,$2,$3) ; }
        | TRANFILE namclause dbclause posclause
                = { $$ = pstree3(TRANFILE,$2,$3,$4) ; }
        | TRANFILE namclause dbclause keyclause
                = { $$ = pstree3(TRANFILE,$2,$3,$4) ; }
        | TRANFILE namclause dbclause posclause keyclause
                = { $$ = pstree4(TRANFILE,$2,$3,$4,$5) ; }
        | FIELD fnmclause
                = { $$ = pstree1(FIELD,$2) ; }
        | FIELD nam2clause POS '(' INTEGER ')'
                = { $$ = pstree2(FIELD,$2,$5) ; }
        | FIELD nam2clause POS '(' INTEGER ',' INTEGER ')'
                = { $$ = pstree3(FIELD,$2,$5,$7) ; }
        | RGROUP nmclause
                = { $$ = pstree1(RGROUP,$2) ; }
        | ARRAY nmclause
                = { $$ = pstree1(ARRAY,$2) ; }
        | ETRAN                         /* ETRAN statement */
                = { $$ = pstreeno(ETRAN) ; } ;
namclause:
        NAME
        | nmclause;
nmclause:
        NAME '(' CHARCONS ')'
                = { $$ = pstree2(SNAME,$1,$3) ; } ;
fnmclause:
        nmclause
        | nm2clause ;
```

```
nam2clause:
        namclause
        | nm2clause ;
nm2clause:
        NAME '(' CHARCONS ',' CHARCONS ')'
                = { $$ = pstree3(SNAME,$1,$3,$5) ; } ;
dbclause:
        DATABASE '(' NAME ')'
                = { $$ = pstree1(DATABASE,$3) ; } ;
posclause:
        POSITION '(' INTEGER ')'
                = { $$ = pstree1(POSITION,$3) ; } ;
keyclause:
        KEY '(' NAME ')'
                = { $$ = pstree1(KEY,$3) ; } ;
```

# APPENDIX B

## ADAPT I FILE DICTIONARY AND RECORD MAP UTILIZATION

This appendix provides illustrative material which will better describe how ADAPT I stores and utilizes a target system file's description and internal data records for that file. Figure B-1 shows the logical structure of a fictitious PERSONNL file as it would be visualized by ADAPT I. This file contains fields describing each employee's present status, family situation, and past job experience. Each employee's basic data set (BDS) consists of his first and last names, his title and salary, and, if married, his/her spouse's first name. Subordinate to the basic data set are two repeating groups, one describing children and one describing past job experience. Each occurrence of the CHILDREN repeating group describes a child and consists of the child's first name and hobbies. Each occurrence of the EXPERNCE repeating group describes a previous job and consists of the company's name and the number of years worked. Subordinate to the EXPERNCE repeating group is another repeating group describing projects worked on. Each occurrence of the PROJECTS repeating group describes a project and consists of the project name and the supervisor's name. In order to go from the logical file structure in figure B-1 to the file dictionary data shown in figure B-2, a DDL SCHEMA run must be input to ADAPT I.

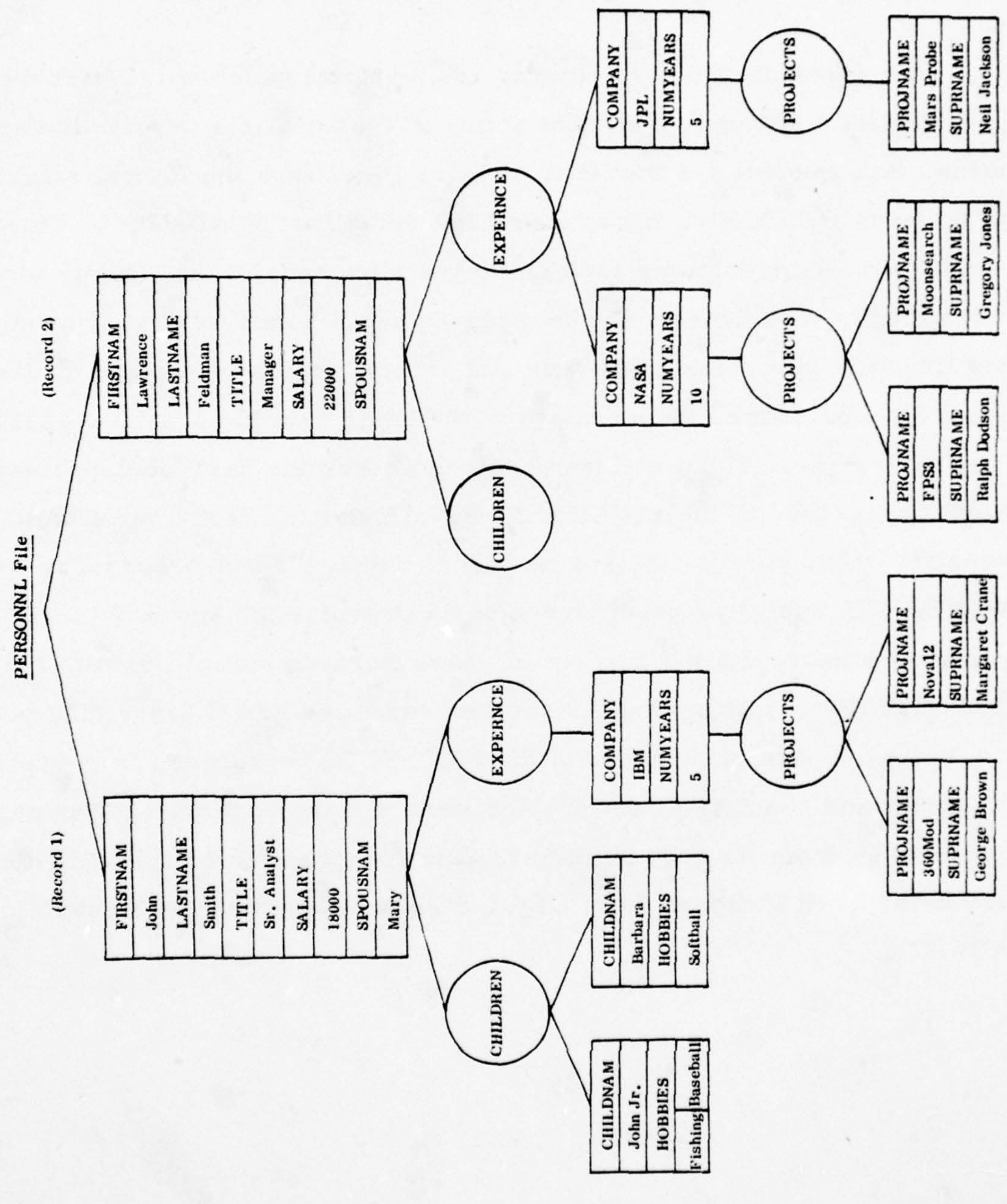
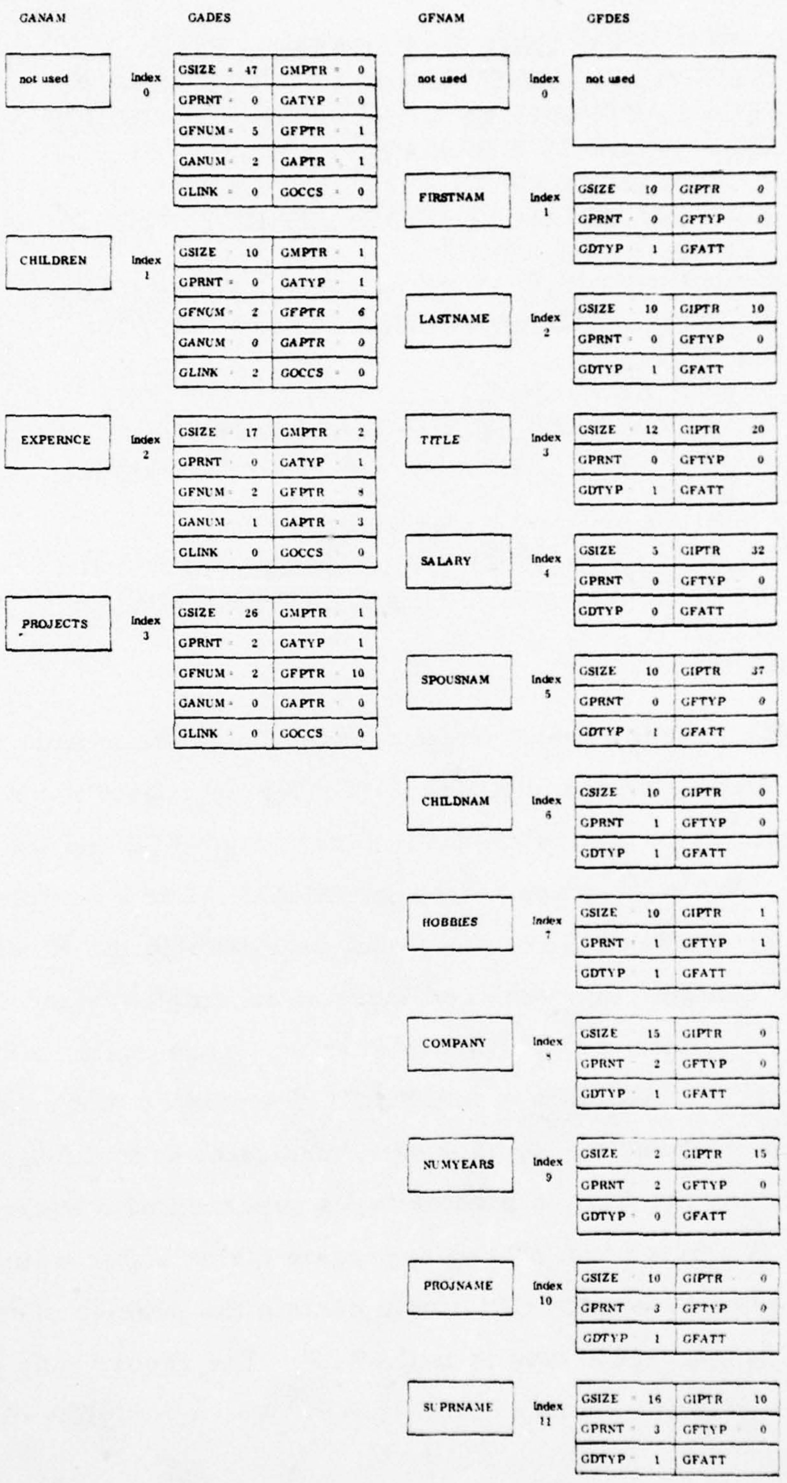


Figure B-1. UDL File.

The input for the PERSONNL file is as follows:

```
SCHEMA PERSONNL DATABASE(QLP);
FIELD FIRSTNAM SV CHAR ATT(KEY, VIS) 10;
FIELD LASTNAME SV CHAR ATT.(KEY, VIS) 10;
FIELD TITLE SV CHAR ATT(KEY, VIS) 12;
FIELD SALARY SV NUM ATT(KEY, VIS) 5;
FIELD SPOUSNAM SV CHAR ATT(KEY, VIS) 10;
RGROUP CHILDREN;
FIELD CHILDNAM SV CHAR ATT(KEY, VIS) 10;
FIELD HOBBIES MV CHAR ATT(KEY, VIS) 10;
ERGROUP;
RGROUP EXPERNCE;
FIELD COMPANY SV CHAR ATT(KEY, VIS) 15;
FIELD NUMYEARS SV NUM ATT(KEY, VIS) 2;
ERGROUP;
RGROUP PROJECTS EXPERNCE;
FIELD PROJNAME SV CHAR ATT(KEY, VIS) 10;
FIELD SUPRNAME SV CHAR ATT(KEY, VIS) 16;
ERGROUP;
ESCHEMA;
```

The DDL process reads and processes these statements and builds the dictionary files as shown in figure B-2. The aggregate names (GANAM) and aggregate descriptions (GADES) describe the BDS and the three repeating groups. The names are stored in GANAM. For a complete description of the data stored in GADES, the global data description should be read. The GSIZE field contains the combined sizes of all single-valued fields in the aggregate. A pointer to the first field in the aggregate is in GFPTR and the number of sequential fields is in GFNUM. A pointer to a subordinate aggregate is in GAPTR and the number of subordinates is in GANUM. For aggregates other than the BDS, a pointer to its superordinate aggregate is in GPRNT and a pointer to a sibling aggregate (same superordinate) is in GLINK. For an array, GOCCS would contain the number of occurrences allowed. The aggregate type is in GATYP. The record map pointer in GMPTR is a relative pointer into GRMAP. For a complete discussion of



| GANAM | | GADES | |
|---|---|---|---|
| not used | Index 0 | GSIZE = 47 | GMPTR = 0 |
| | | GPRNT = 0 | GATYP = 0 |
| | | GFNUM = 5 | GFPTR = 1 |
| | | GANUM = 2 | GAPTR = 1 |
| | | GLINK = 0 | GOCCS = 0 |
| CHILDREN | Index 1 | GSIZE 10 | GMPTR 1 |
| | | GPRNT = 0 | GATYP 1 |
| | | GFNUM 2 | GFPTR 6 |
| | | GANUM 0 | GAPTR 0 |
| | | GLINK 2 | GOCCS 0 |
| EXPERNCE | Index 2 | GSIZE 17 | GMPTR 2 |
| | | GPRNT 0 | GATYP 1 |
| | | GFNUM 2 | GFPTR 8 |
| | | GANUM 1 | GAPTR 3 |
| | | GLINK 0 | GOCCS 0 |
| PROJECTS | Index 3 | GSIZE 26 | GMPTR 1 |
| | | GPRNT 2 | GATYP 1 |
| | | GFNUM 2 | GFPTR 10 |
| | | GANUM 0 | GAPTR 0 |
| | | GLINK 0 | GOCCS 0 |

| GFNAM | | GFDES | |
|---|---|---|---|
| not used | Index 0 | not used | |
| FIRSTNAM | Index 1 | GSIZE 10 | GIPTR 0 |
| | | GPRNT 0 | GFTYP 0 |
| | | GDTYP 1 | GFATT |
| LASTNAME | Index 2 | GSIZE 10 | GIPTR 10 |
| | | GPRNT 0 | GFTYP 0 |
| | | GDTYP 1 | GFATT |
| TITLE | Index 3 | GSIZE 12 | GIPTR 20 |
| | | GPRNT 0 | GFTYP 0 |
| | | GDTYP 1 | GFATT |
| SALARY | Index 4 | GSIZE 5 | GIPTR 32 |
| | | GPRNT 0 | GFTYP 0 |
| | | GDTYP 0 | GFATT |
| SPOUSNAM | Index 5 | GSIZE 10 | GIPTR 37 |
| | | GPRNT 0 | GFTYP 0 |
| | | GDTYP 1 | GFATT |
| CHILDNAM | Index 6 | GSIZE 10 | GIPTR 0 |
| | | GPRNT 1 | GFTYP 0 |
| | | GDTYP 1 | GFATT |
| HOBBIES | Index 7 | GSIZE 10 | GIPTR 1 |
| | | GPRNT 1 | GFTYP 1 |
| | | GDTYP 1 | GFATT |
| COMPANY | Index 8 | GSIZE 15 | GIPTR 0 |
| | | GPRNT 2 | GFTYP 0 |
| | | GDTYP 1 | GFATT |
| NUMYEARS | Index 9 | GSIZE 2 | GIPTR 15 |
| | | GPRNT 2 | GFTYP 0 |
| | | GDTYP 0 | GFATT |
| PROJNAME | Index 10 | GSIZE 10 | GIPTR 0 |
| | | GPRNT 3 | GFTYP 0 |
| | | GDTYP 1 | GFATT |
| SUPRNAME | Index 11 | GSIZE 16 | GIPTR 10 |
| | | GPRNT 3 | GFTYP 0 |
| | | GDTYP 1 | GFATT |

Figure B-2. Dictionary Data for PERSONNL File.

GRMAP and its two types of nodes, global data should be consulted. The data set node for the basic data set starts, by convention, in index zero of GRMAP. Each aggregate subordinate to the BDS, then, has a GMPTR value which is relative to index zero. Figure B-3 shows actual internal records which would be built using the data shown in figure B-1 and the file dictionary data shown in figure B-2. For example, repeating group EXPERNCE has a GMPTR value of 2. Therefore, word index 2 of GRMAP contains a pointer to an occurrence node for EXPERNCE. In the case of Record 1, this pointer is 12 and at index 12 is found an occurrence node showing that EXPERNCE has one occurrence and that that occurrence is at index 14. At index 14 is a data set node for EXPERNCE showing that data for the single-valued fields start at index 98 in GRDAT. Subordinate to EXPERNCE is repeating group PROJECTS with a GMPTR value of 1. This is relative to the base of a data set node of EXPERNCE, in this case index 14. Therefore, in word index 15 is found a pointer to an occurrence node for PROJECTS.

DDL stores field names in GFNAM and field descriptive data in GFDES. Read the global data descriptions for a complete description of GFDES. The GSIZE field contains the maximum number of characters for a fixed size field and represents the actual amount of space saved in GRDAT for that field's data. For a multivalued or variable length field, GIPTR has the same use as GMPTR in GADES. It is a relative pointer into GRMAP, in this case to two words containing a pointer into GRDAT and either a character count or an occurrence count. For example, repeating group CHILDREN has a multivalued field HOBBIES. CHILDREN is subordinate to the BDS and has a GMPTR value of 1. At index 1 (for Record 1) is a pointer to an occurrence node at index 3. The occurrence node indicates two occurrences, at indexes 6 and 9. At index 6 is a data set node for CHILDREN. Field HOBBIES has a GIPTR of 1; therefore, at index 7 is a pointer into GRDAT (index 68) and at index 8 is the occurrence count.

Record 1

GRMAP

| Index | GRMAP |
|---|---|
| 0 | (1) |
| 1 | 3 |
| 2 | 12 |
| 3 | 2 |
| 4 | 6 |
| 5 | 9 |
| 6 | (48) |
| 7 | (68) |
| 8 | 2 |
| 9 | (58) |
| 10 | (88) |
| 11 | 1 |
| 12 | 1 |
| 13 | 14 |
| 14 | (98) |
| 15 | 16 |
| 16 | 2 |
| 17 | 19 |
| 18 | 20 |
| 19 | (115) |
| 20 | (141) |

GRDAT

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | – | J | O | H | N | △ | △ | △ | △ | △ |
| 10 | △ | S | M | I | T | H | △ | △ | △ | △ |
| 20 | △ | S | R | . | △ | A | N | A | L | Y |
| 30 | S | T | △ | 1 | 8 | 0 | 0 | 0 | M | A |
| 40 | R | Y | △ | △ | △ | △ | △ | △ | J | O |
| 50 | H | N | △ | J | R | . | △ | △ | B | A |
| 60 | R | B | A | R | A | △ | △ | △ | F | I |
| 70 | S | H | I | N | G | △ | △ | △ | B | A |
| 80 | S | E | B | A | L | L | △ | △ | S | O |
| 90 | F | T | B | A | L | L | △ | △ | I | B |
| 100 | M | △ | △ | △ | △ | △ | △ | △ | △ | △ |
| 110 | △ | △ | △ | 5 | △ | 3 | 6 | 0 | M | O |
| 120 | D | △ | △ | △ | △ | G | E | O | R | G |
| 130 | E | △ | B | R | O | W | N | △ | △ | △ |
| 140 | △ | N | O | V | A | I | 2 | △ | △ | △ |
| 150 | △ | M | A | R | G | A | R | E | T | △ |
| 160 | C | R | A | N | E | △ | △ |  |  |  |

Record 2

GRMAP

| Index | GRMAP |
|---|---|
| 0 | (1) |
| 1 | 0 |
| 2 | 3 |
| 3 | 2 |
| 4 | 6 |
| 5 | 13 |
| 6 | (48) |
| 7 | 8 |
| 8 | 2 |
| 9 | 11 |
| 10 | 12 |
| 11 | (65) |
| 12 | (91) |
| 13 | (117) |
| 14 | 15 |
| 15 | 1 |
| 16 | 17 |
| 17 | (134) |

GRDAT

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | – | L | A | W | R | E | N | C | E | △ |
| 10 | △ | F | E | L | D | M | A | N | △ | △ |
| 20 | △ | M | A | N | A | G | E | R | △ | △ |
| 30 | △ | △ | △ | 2 | 2 | 0 | 0 | 0 | △ | △ |
| 40 | △ | △ | △ | △ | △ | △ | △ | △ | N | A |
| 50 | S | A | △ | △ | △ | △ | △ | △ | △ | △ |
| 60 | △ | △ | △ | 1 | 0 | F | P | S | 3 | △ |
| 70 | △ | △ | △ | △ | △ | R | A | L | P | H |
| 80 | △ | D | O | D | S | O | N | △ | △ | △ |
| 90 | △ | M | O | O | N | S | E | A | R | C |
| 100 | H | G | R | E | G | O | R | Y | △ | J |
| 110 | O | N | E | S | △ | △ | △ | J | P | L |
| 120 | △ | △ | △ | △ | △ | △ | △ | △ | △ | △ |
| 130 | △ | △ | 5 | △ | M | A | R | S | △ | P |
| 140 | R | O | B | E | N | E | I | L | △ | J |
| 150 | A | C | K | S | O | N | _ | △ | △ | △ |

Values in parentheses are pointers to GRDAT

Figure B-3. Record Data for PERSONNL File.

For a single-valued field, GIPTR is a relative pointer into GRDAT. For example, field LASTNAME in the BDS has a GIPTR of 10. The dataset node for the BDS starts in word zero of GRMAP and indicates that data for the BDS start in index 1 of GRDAT. Field LASTNAME's value is stored relative to 1; i.e., at index 11. Also in GFDES, GPRNT contains a pointer to the aggregate which contains the field. GFTYP contains the field's type, either single-valued, multivalued, or variable length. GDTYP specifies the field's data type, either character, numeric, or geographic. GFATT contains the interrogation and display attributes of the field.

In the two records shown in figure B-3, the record maps are relatively small. For a file which is defined with no repeating groups or arrays, the record map would consist only of the data set node for the BDS and would likely be even smaller. A large number of aggregates and nested aggregates could cause the record maps to become large.

# APPENDIX C

# TRANSFORMATION DECOMPILING TABLES

This appendix presents the transformation decompiling tables for ADAPT I. As described under a separate discussion of the ADAPT I decompiler (TIDECOMP), all transformations from UDL to a given target host language are accomplished through a set of data structures, TIPRM, TITRG, and TIACT.

Each TSIG process (i. e. , TSIG1 for RYETIP, TSIG2 for DIAOLS, TSIG3 for SOLIS, and TSIG4 for QLP) contains the same decompiler and the appropriate set of decompiling tables. Below is a graphic representation of the decompiling tables utilized for all four target host transformations in ADAPT I. For completeness, the basic decompiling process is sketched below.

## Decompiling Process

Each invocation of the decompiler is passed an absolute index into a parse tree (GTREE). This index always points to the head of a node, hence always points to some UDL token. The decompiler searches the primitive token array, TIPRM, for a match against the specified token. Upon a match in TIPRM the decompiler uses the next two array entries in TIPRM as indices into the transformation string array, TITRG, and the decompiling action array, TIACT, respectively. At this point, the decompiler processes each character in TITRG until it reaches a null character which signifies the completion of this decompiling pass. Most decompiling sequences involve many recursive invocations of TIDECOMP. The decompiler operates entirely against the substring in TITRG and all actions are controlled by special characters that may appear in the substring. The decompiler recognizes two special characters, (three, if the terminating null character is included), the backslash '\' and the caret '^'. All other characters are output as part of the transformation (including blanks).

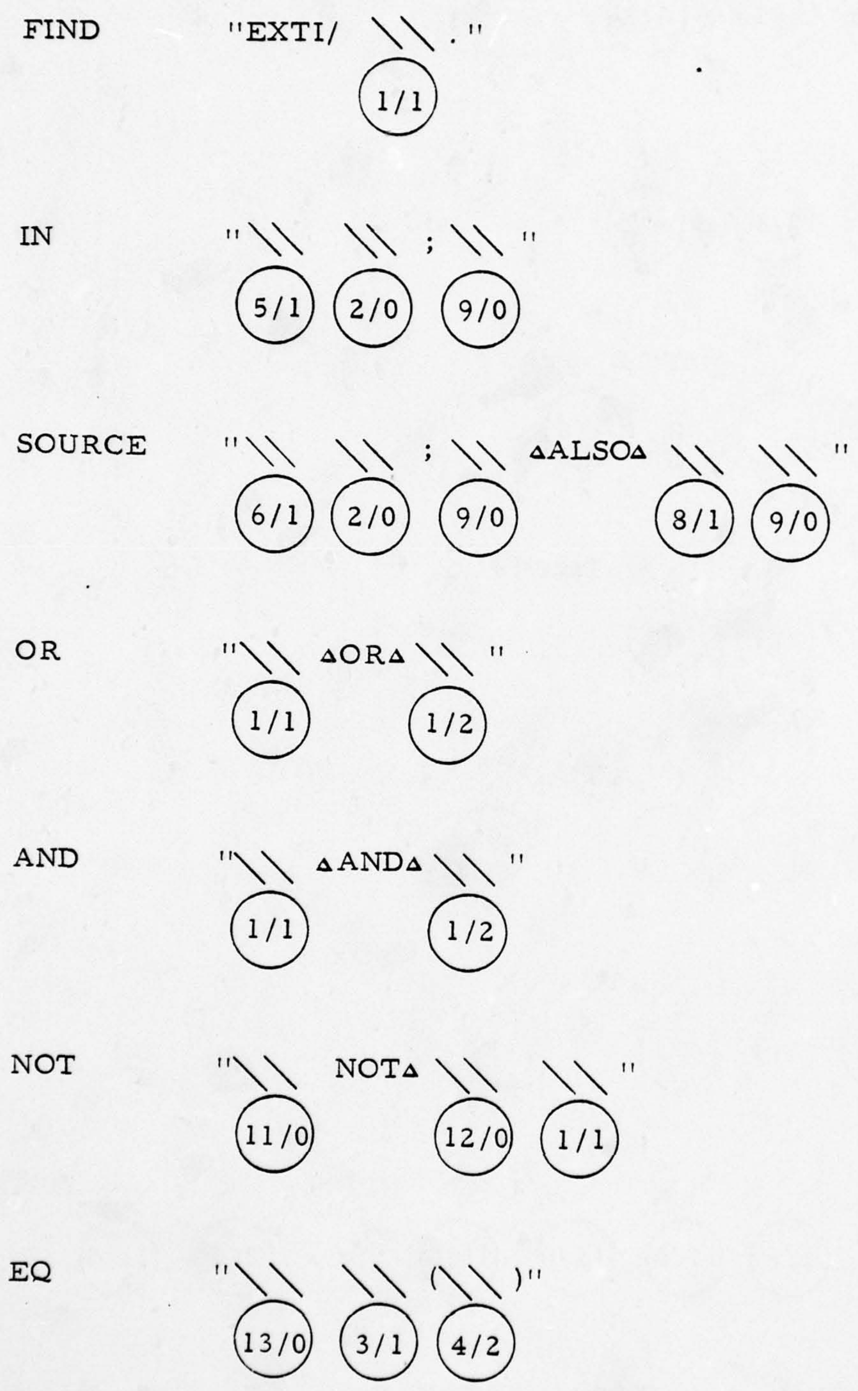
The backslash character indicates to the decompiler that a special decompiling action function must be executed. The current position in the decompiling action array (whose initial starting position is pointed to by TIPRM) points to the appropriate decompiling function, and the next adjacent entry is a relative pointer to the leg of the current parse tree node which is to be passed as input to the decompiling function. An entry of zero implies no input is required for this decompiling function. The caret character indicates that the current UDL token is a list node and that all processing is to be controlled by the leg count contained in the node. The caret character is followed by two special control characters. The first specifies the starting node leg and the second specifies the number of list-separators required for output. The indicated list-separators directly follow the two special control characters.

## Decompiling Tables

For legibility, the four decompiling tables have been represented graphically. A few comments are required on their form. For each decompiling set (per host system) the appropriate token is presented on the left side as it is actually defined in ADAPT I (i. e., the C DEFINE statement representation of the token). Following the symbolic token representation is the actual transformation substring as found in TITRG (surrounded by double quotes) that is pointed to by TIPRM for this token. Immediately below each special backslash character (represented as a pair due to UNIX escape character conventions) are the decompiling function number and node leg pointers. Each pair is circled for easy identification. The first number is an index into a function switch and the second number is a pointer to the appropriate node leg for input to the action function. These number pairs constitute entries in the integer array TIACT.

For convenience, a list of the special decompiling action functions used in ADAPT I are provided below following the decompiling tables. The function number corresponds to the first number within a circled pair.

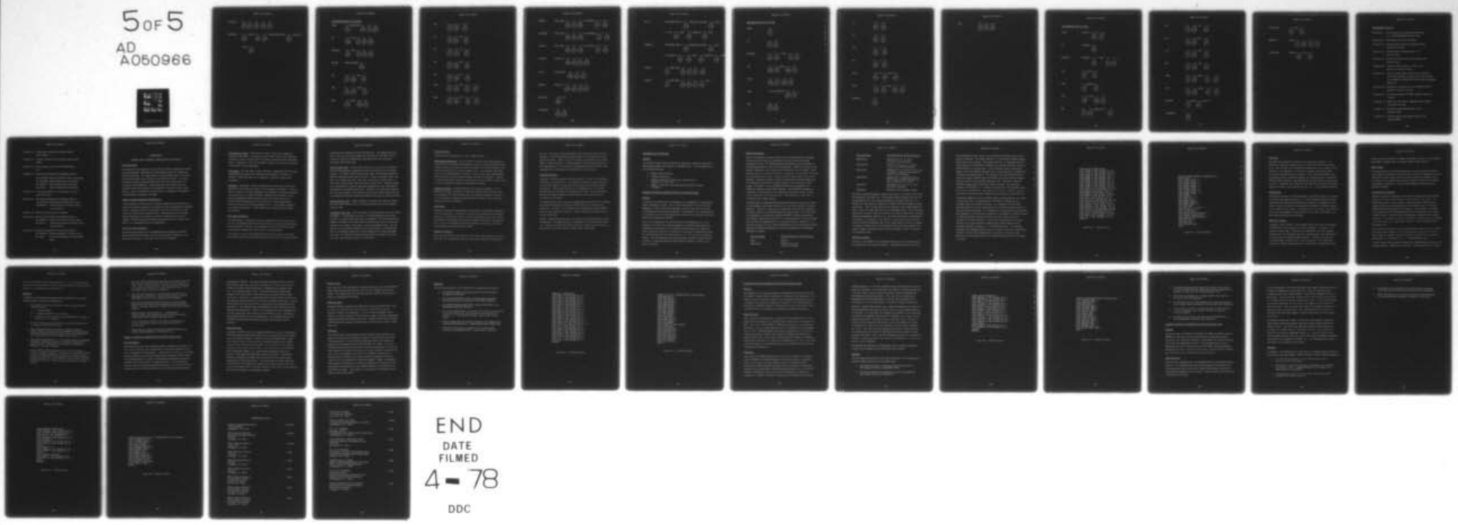
## Decompiling Tables for RYETIP


FIND "EXTI/ \\ ." 1/1
IN "\\ \\ ; \\ " 5/1 2/0 9/0
SOURCE "\\ \\ ; \\ ΔALSOΔ \\ \\ " 6/1 2/0 9/0 8/1 9/0
OR "\\ ΔORΔ \\ " 1/1 1/2
AND "\\ ΔANDΔ \\ " 1/1 1/2
NOT "\\ NOTΔ \\ \\ " 11/0 12/0 1/1
EQ "\\ \\ (\\ )" 13/0 3/1 4/2

```
LT      "\\    \\ <( \\ )"
        (13/0) (3/1) (4/2)

GT      " \\   \\ >( \\ )"
        (13/0) (3/1) (4/2)

LE      "\\    \\ (*ΔTOΔ\\)"
        (13/0) (3/1)      (4/2)

GE      " \\    \\ ( \\ΔTOΔ*)"
        (13/0) (3/1) (4/2)

WRG     "\\    \\ ( \\ΔTOΔ  \\)"
        (13/0) (3/1) (4/2)    (4/3)

ORG     " \\   \\ (\\ΔTOΔ\\)"
        (14/0) (3/1) (4/2)  (4/3)

SAVE    "\\   \\   \\    \\    \\ PRINT/ \\ ; \\ OPTION:
        (6/1) (8/1) (17/0) (18/0) (15/0)    (2/0)  (15/0)

          SPACE(1) \\ FORMAT:NONE."
                  (15/0)
```

AD-A050 966 LOGICON INC SAN DIEGO CALIF F/G 5/2
ADAPT I FINAL FUNCTIONAL AND SYSTEM DESIGN SPECIFICATION.(U)
JAN 78 L R ERICKSON, M E SOLEGLAD N00014-76-C-0899
UNCLASSIFIED 76-C-0899-2-F NL

5 OF 5
AD
A050966



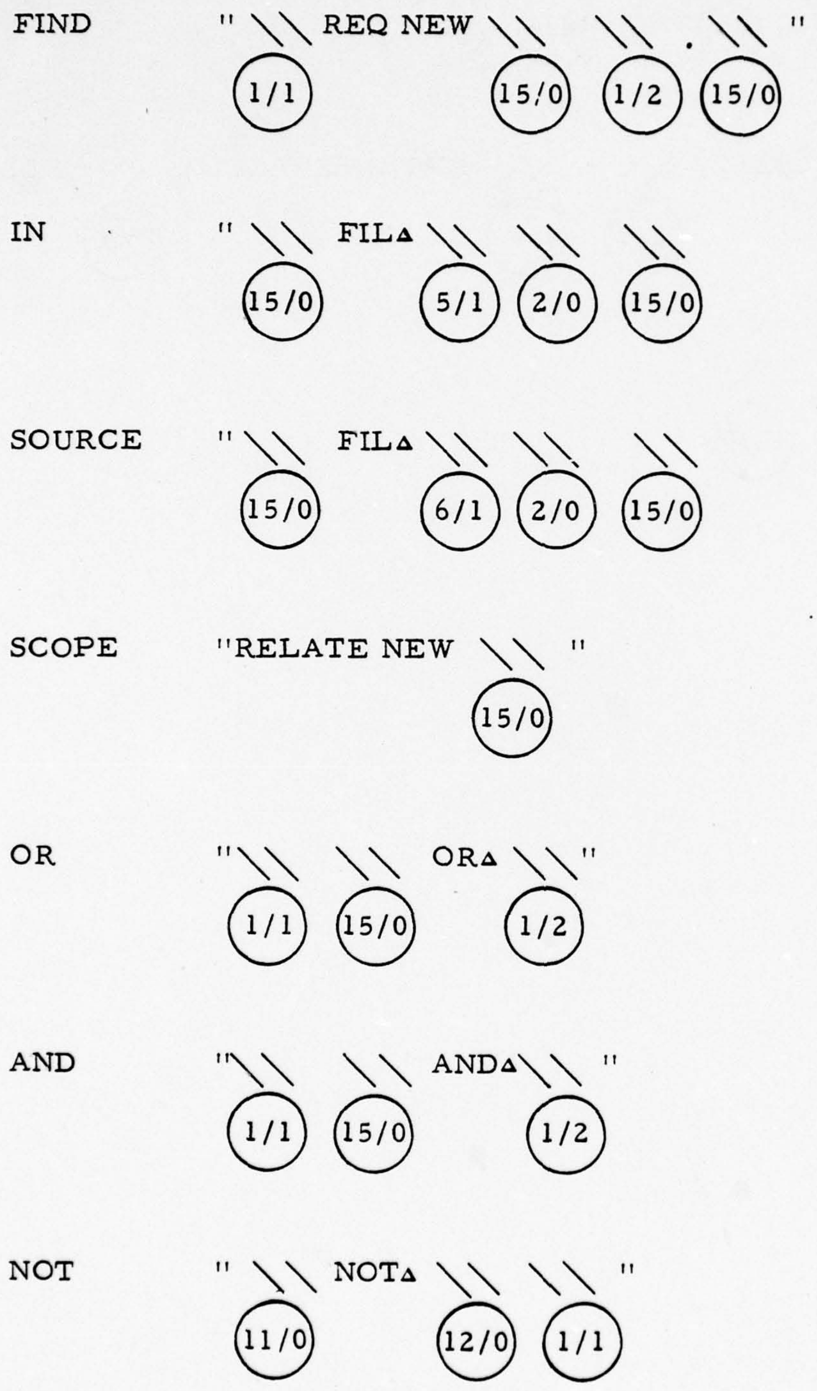
END
DATE
FILMED
4-78
DDC

DISPLAY "$\underset{6/1}{\backslash\backslash}$ $\underset{8/1}{\backslash\backslash}$ $\underset{17/0}{\backslash\backslash}$ $\underset{18/0}{\backslash\backslash}$ $\underset{16/2}{\backslash\backslash}$ ."

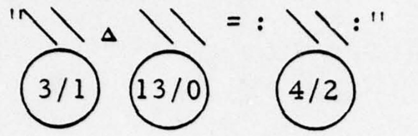
DISPLIST "$\underset{15/0}{\backslash\backslash}$ PRINT/ $\underset{2/0}{\backslash\backslash}$ ; $\underset{15/0}{\backslash\backslash}$ OPTION:SPACE(1) $\underset{15/0}{\backslash\backslash}$ FORMAT:

PART $\underset{15/0}{\backslash\backslash}$ "

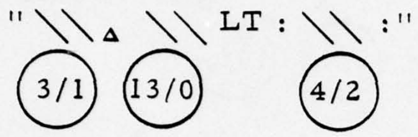
Decompiling Tables for DIAOLS



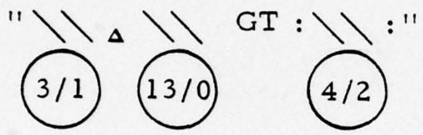
EQ



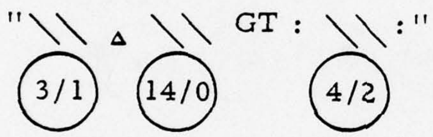
LT



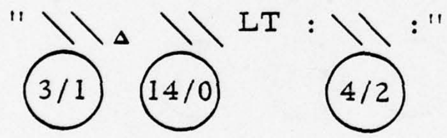
GT



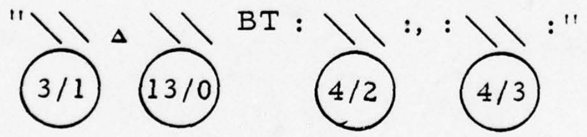
LE



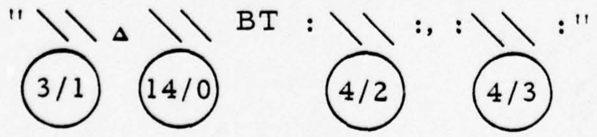
GE



WRG



ORG

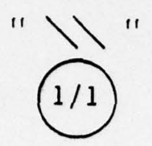


```
INSIDE       "GEO NEW  \\     \\     \\   IF INSIDE△  \\    R   \\   "
                      (15/0) (3/1) (15/0)             (1/2)     (15/0)

OUTSIDE      "GEO NEW  \\     \\     \\   IF OUTSIDE△  \\    R   \\   "
                      (15/0) (3/1) (15/0)              (1/2)     (15/0)

ALONG        "GEO NEW  \\     \\     \\   IF ALONG△  \\    R   \\   "
                      (15/0) (3/1) (15/0)            (1/2)     (15/0)

CIRCLE       "CIRCLE-1  \\     \\ ,  \\     \\   "
                       (15/0) (4/2) (4/1) (15/0)

POLY         "POLYGON-1  \\     \\    \\   "
                        (15/0) (21/1) (1/1)

ROUTE        "ROUTE-1  \\     \\ ,   \\    \\   "
                      (15/0) (21/2) (4/1) (1/2)

GEOLIST      " \\^\\1\0"
             (15/0)

GEOCONS      " \\    \\  "
             (4/1) (15/0)
```

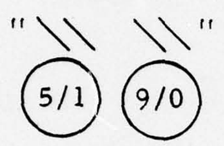
```
SAVE        "COMPOSE NEW, V \\ HEADING EQ ESN. \\ DATA
                           (15/0)              (15/0)

            Δ = ALL. \\ END. \\ DISPLAY \\ BYE"
                   (15/0) (15/0)     (15/0)

DISPLAY     "COMPOSE NEW, V \\ HEADING EQ ESN. \\ DATA
                           (15/0)              (15/0)

            Δ = SELECT Δ \\ Δ . \\ END. \\ DISPLAY \\ BYE"
                       (16/2)  (15/0) (15/0)    (15/0)

FINDD1      " \\ REQ NEW \\   \\   \\ .  \\ "
            (1/1)      (15/0)(8/1)(9/0) (15/0)

FINDD2      " \\ REQ NEW \\   \\ ( \\ ) \\  AND ("
            (1/1)      (15/0)(8/1) (9/0) (15/0)
```

<u>Decompiling Tables for SOLIS</u>

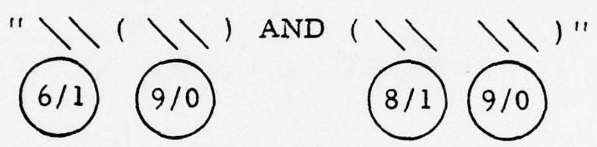
FIND



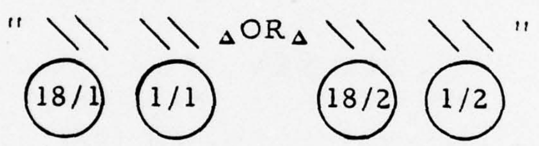
IN



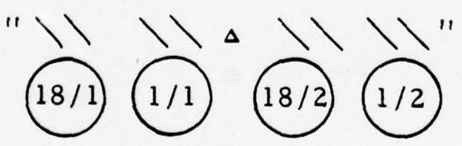
SOURCE



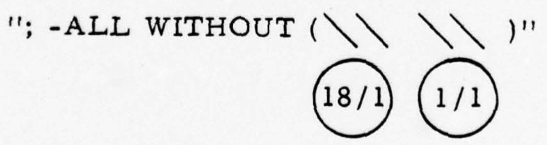
OR



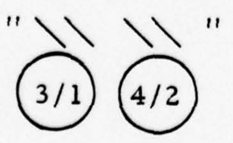
AND



NOT

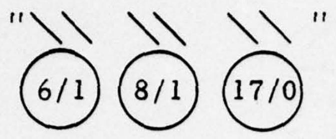


EQ



| | |
|---|---|
| LT | $'' \underset{\textcircled{\scriptsize 16/1}}{\backslash\backslash} < \underset{\textcircled{\scriptsize 4/2}}{\backslash\backslash} ''$ |
| GT | $'' \underset{\textcircled{\scriptsize 16/1}}{\backslash\backslash} > \underset{\textcircled{\scriptsize 4/2}}{\backslash\backslash} ''$ |
| LE | $'' \underset{\textcircled{\scriptsize 16/1}}{\backslash\backslash} <= \underset{\textcircled{\scriptsize 4/2}}{\backslash\backslash} ''$ |
| GE | $'' \underset{\textcircled{\scriptsize 16/1}}{\backslash\backslash} >= \underset{\textcircled{\scriptsize 4/2}}{\backslash\backslash} ''$ |
| WRG | $'' \underset{\textcircled{\scriptsize 16/1}}{\backslash\backslash}\ R= \underset{\textcircled{\scriptsize 4/2}}{\backslash\backslash}\ \Delta\text{TO}\Delta \underset{\textcircled{\scriptsize 4/3}}{\backslash\backslash} ''$ |
| ORG | $''( \underset{\textcircled{\scriptsize 16/1}}{\backslash\backslash} < \underset{\textcircled{\scriptsize 4/2}}{\backslash\backslash}\ \Delta\text{OR}\Delta \underset{\textcircled{\scriptsize 16/1}}{\backslash\backslash} > \underset{\textcircled{\scriptsize 4/3}}{\backslash\backslash} )''$ |
| DISPLAY | $'' \underset{\textcircled{\scriptsize 1/1}}{\backslash\backslash} ''$ |

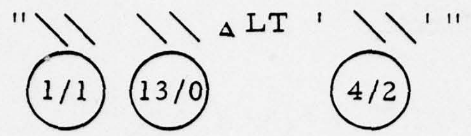
TAG



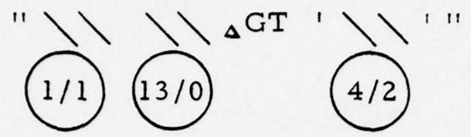
Decompiling Tables for QLP

| | |
|---|---|
| FIND | "COUNT $\triangle$ $\underset{\textcircled{20/1}}{\backslash\backslash}$ $\triangle$ $\underset{\textcircled{1/1}}{\backslash\backslash}$" |
| IN | "WHERE $\triangle$ $\underset{\textcircled{9/0}}{\backslash\backslash}$" |
| SOURCE | "WHERE ( $\underset{\textcircled{9/0}}{\backslash\backslash}$ ) AND ( $\underset{\textcircled{8/1}}{\backslash\backslash}$ $\underset{\textcircled{9/0}}{\backslash\backslash}$ )" |
| OR | " $\underset{\textcircled{1/1}}{\backslash\backslash}$ $\triangle$ OR $\triangle$ $\underset{\textcircled{1/2}}{\backslash\backslash}$ " |
| AND | " $\underset{\textcircled{1/1}}{\backslash\backslash}$ $\triangle$ AND $\triangle$ $\underset{\textcircled{1/2}}{\backslash\backslash}$ " |
| NOT | " $\underset{\textcircled{11/0}}{\backslash\backslash}$ $\triangle$ NOT $\underset{\textcircled{12/0}}{\backslash\backslash}$ $\underset{\textcircled{1/1}}{\backslash\backslash}$" |
| EQ | "$\underset{\textcircled{1/1}}{\backslash\backslash}$ $\underset{\textcircled{13/0}}{\backslash\backslash}$ $\triangle$ EQUAL TO ' $\underset{\textcircled{4/2}}{\backslash\backslash}$ ' " |

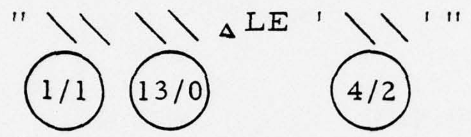
LT



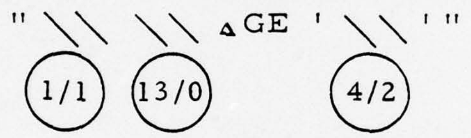
GT



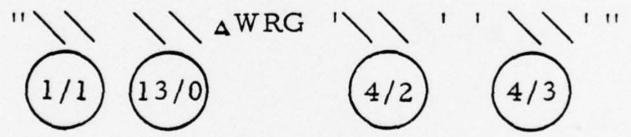
LE



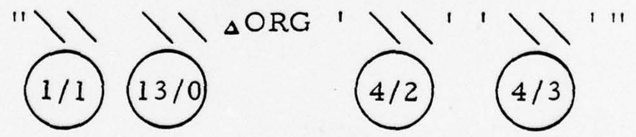
GE



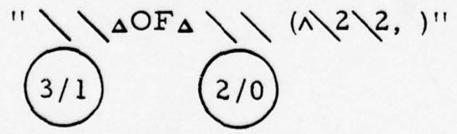
WRG



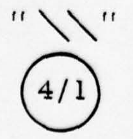
ORG



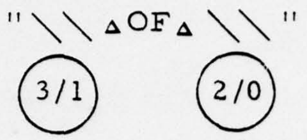
SNAME



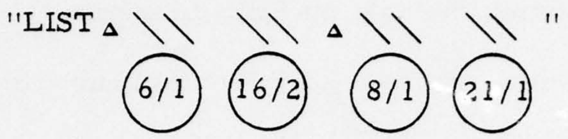
INTEGER



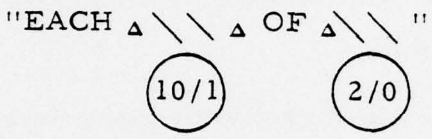
FLDNAME



DISPLAY



AGGNAME



Decompiling Functions

| | |
|---|---|
| Function 01: | Traverse parse tree through indicated leg (i. e., call TIDECOMP with new leg). |
| Function 02: | Output the host specific filename. |
| Function 03: | Output the host specific field name pointed to by parse tree leg. |
| Function 04: | Output the data constant pointed to by parse tree leg. |
| Function 05: | Retrieve FLID directly for file pointed to by parse tree leg. |
| Function 06: | Retrieve FLID indirectly via tag for file pointed to by parse tree leg. |
| Function 07: | Read in all TDL tables required for transformations of file pointed to by input FLID (this function is not called by TIDECOMP but by other decompiling functions). |
| Function 08: | Establish a new parse tree set (GTREE/GTDAT) pointed to by parse tree leg. |
| Function 09: | Force decompilation of FIND statement selection criteria. |
| Function 10: | Output the host specific aggregate name pointed to by parse tree leg. |
| Function 11: | Set logical string buffer pointer to the alternate buffer. |
| Function 12: | Restore logical string buffer pointer to the standard buffer. |

| | | |
|---|---|---|
| Function 13: | Output data contained in alternate logical string buffer. | |
| Function 14: | Remove contents from alternate logical string buffer. | |
| Function 15: | Force output of the current transformation line. | |
| Function 16: | Special host dependent decompiling function. | |
| | 16r (RYETIP): | Special FORMAT: PART processing. |
| | 16d (DIAOLS): | Special display-list processing. |
| | 16s (SOLIS): | Special field name processing. |
| | 16q (QLP): | Special display-list processing. |
| Function 17: | Force decompilation of a statement (from the head of the tree). | |
| Function 18: | Special host dependent decompiling function. | |
| | 18r (RYETIP): | Reestablish original parse trees. |
| | 18s(SOLIS): | Protect against rel-term overlap on screen line. |
| Function 19: | Special null function (does nothing). | |
| Function 20: | Special host dependent decompiling function. | |
| | 20d (DIAOLS): | Outputs special query line number. |
| | 20q (QLP): | Performs special processing for tag/ file references. |
| Function 21: | Special host dependent decompiling function. | |
| | 21d (DIAOLS): | Outputs geographic constant counts. |
| | 21q (QLP): | Forces decompiling of a IN/SOURCE token. |

# APPENDIX D

# SCHEMA AND TRANFILE GENERATION FOR ADAPT I

## INTRODUCTION

The purpose of this appendix is to provide the ADAPT I superuser with the rules governing the specification of transformation data via tranfile and schema definitions. This specification of transformation data is governed by rigid rules which may, however, vary from host to host. The rules themselves and the reasoning behind each rule will be discussed in the remainder of this appendix. These discussions are broken down into sections by host and include a brief summary of rules at the end of each section. Prior to individual host discussions, a summary of the general transformation technology is provided.

## BASIC TRANSFORMATION TECHNOLOGY

A large number of the rules and conventions adopted for the generation of schema and tranfile sequences are dictated by the transformation technology itself. A smaller subset is host specific and will be covered in the individual host discussions presented below. Although much of the transformation technology is discussed in the UDL Transformation section of ADAPT I Final Functional and System Design Specification, Report 76-C-0899-2, it is capsulized here for completeness.

### Data Structure Components

Each host data element for every COINS II file recognized by ADAPT I must be analyzed with respect to three data structure components: the data structure type, its data type, and its set of attributes.

Data Structure Types. Data structure types in UDL are grouped into aggregates and fields. The former can be divided further into repeating groups and arrays, and the latter into single-valued fields and multivalued fields. Aggregates, which can be multioccurring, form named collections of fields and/or other aggregates.

Data Types. The data type of a data structure is applicable for fields only and specifies the kind of data which the field contains. Currently, in ADAPT I, three data types are recognized: character, numeric and geographic.

Attributes. Attributes, which are applicable for fields only, specify the operability of the field with respect to interrogation and display. Interrogation attributes are keyed, dependent and nonkeyed. Keyed data elements can be queried in any FIND statement, dependent data elements can be queried in dependent FIND statements only (those which reference a previous FIND statement for its record source), and nonkeyed fields cannot be referenced in FIND statements at all. Valid display attributes are visible and invisible. Visible fields can be displayed and invisible fields cannot.

## Data Type Assignment

The specification of valid data structure mappings from a host data structure to a recognizable UDL data structure will be covered in the individual host discussions. However, the assignment of data types and attributes is essentially independent of the host and is discussed here.

The primary concern in assigning data types for UDL fields is ensuring that the particular target host does indeed provide a set of constructs which comply with the semantics of the UDL data type. The mapping does not have to be one-to-one (and usually isn't) but the host must have a set of constructs which are semantically equivalent to the UDL constructs concerned with the data type.

Numeric Data Type. Numeric data types have a very precise meaning in UDL and this semantic interpretation must be preserved in the transformations. To assign a field a numeric data type, all relations germane to numeric operations must exist in the host. These are equality (EQ), less than (LT), greater than (GT), less than or equal (LE), greater than or equal (GE), within range (WRG) and outside range (ORG). Another mandatory requirement for numeric data type assignments is that only numeric information be contained in the field. Therefore, this eliminates numeric data type assignments to data elements which might contain character data even though the relational operations defined above may be legal in the host.

Character Data Type. Fields assigned a character data type can contain any valid ASCII character and can only be operated upon by the equality operator (EQ).

Geographic Data Type. UDL recognizes a special postional data type called geographic. A geographic constant is a latitude/longitude position pair. In order to be assigned the data type of geographic, the UDL operations of INSIDE, OUTSIDE, and ALONG must be expressable in the host and must operate over irregular route paths, circle and polygonal areas. Merely because a host has data elements which contain latitude or longitude data does not necessarily imply that it can be mapped onto a UDL geographic data type. All operations mentioned above must be possible in this host such that total semantic integrity is maintained.

## Field Attributes

Field attribute assignments are very straightforward.

Interrogation Attributes. If it is possible to query a given data element in some host with an initial FIND statement, then that corresponding UDL field should be given the attribute of keyed. Similarly, if the host imposes a dependent record source for some operation (i. e., any operation that might occur in the FIND statement's selection criteria), then that corresponding UDL field must be given the interrogation attribute of dependent. Finally, if a given data element cannot be referenced in a query (these do exist), then it must be assigned a nonkeyed interrogation attribute.

Display Attributes. Display attributes are as easily assigned. If it is possible to request the output of some data element from a host then that corresponding UDL field should be given the display attribute of visible. If it is not possible to display the data element via the host, then it must be assigned an invisible attribute.

## Field Width

Data type and attribute assignments are specified in the schema sequence of a UDL file definition. The width of a field is also specified in the schema sequence. In general the width of a field is important and can be correctly determined by consulting the appropriate COINS II file guide. This parameter is used by the DISPLAY process of ADAPT I as well as by both the transformation processes TSIG and TSOT.

## Naming Conventions

Each host data element can be given a UDL name (eight characters or less) which does not necessarily match the host specific name as recognized on the host. Of course, the UDL fieldname must be unique across the file. Similarly, the UDL filename assignment does not have to match the host specific filename. Again, it must be unique across ADAPT I and contain eight or fewer characters. Transformations from host data elements to UDL aggregates are not necessarily straightforward and therefore, these mappings will be discussed in the individual host discussions.

## Tranfile Sequences

In general, tranfile sequences are host specific, however, a few general comments can be made here. The primary purpose of tranfile sequences is to provide ADAPT I transformation processes with host/file specific information. This always includes the correspondence of a UDL data element name to the name as recognized by the host. This is important for both the transformations being generated for the host and for the recognition of host data.

A tranfile sequence also supplies the appropriate information for informing the Target System Output Translator (TSOT) of how to recognize host responses. Currently, two recognition mechanisms exist under ADAPT I, data-by-name and data-by-position. These recognition mechanisms will be discussed below.

Finally, tranfile sequences supply the name of a key data element of a file if applicable. This data element is considered to be a major data element (i.e., always present) whose data contents are unique across the file (i.e., each record of the file contains a unique value in this field).

## SCHEMAS AND TRANFILES

### General

The material which follows presents the rules and conventions required for generating schemas and tranfiles for COINS II hosts. The following hosts are considered herein:

a. SOLIS, interactive host.

b. RYETIP, batch host.

c. DIAOLS, batch host (interactive version may be available at a later date).

d. ISSPIC, interactive host (when DMS-1100/QLP is made available).

## SCHEMA/TRANFILE GENERATIONS FOR THE SOLIS HOST

### General

In general, the SOLIS retrieval language is not independent of the databases (files) residing on its host. This is true with respect to the actual query language, as well as with those language constructs that allow the output of information from selected documents. Consequently, the transformations for this host are not as file independent as they are for the other three hosts presently recognized by ADAPT I.

Much of this dependency has been absorbed by establishing a set of conventions to be used in the tranfile definitions for SOLIS transformations. Although the conventions are controlled by tranfile sequences (which are easy to maintain since they are entered by an interactive subsystem), they are depended upon by the transformations (i. e. , software and decompiling tables). The following material presents these conventions which must be complied to rigorously when new UDL files for SOLIS are created or old files are altered.

## SOLIS Organization

SOLIS is a document retrieval system (or more appropriately a message retrieval system) where the main bulk of the data exists as partially formatted text files. The SOLIS database management system presents an interesting problem for data structure mappings. SOLIS is oriented around separate databases, each of which are divided into a set of volumes. Each database has its own structure however, in general, the same type of queries can be applied across them. Volumes apply to a date range within a given database. Different volumes, which are identically structured, differ only in the number of documents contained in them, (as dictated by the date range). In ADAPT I, only two SOLIS databases are recognized: the PRODUCT database and the SIRE database. The PRODUCT database is composed of many volumes, each reflecting a given date range, varying from a single day to sixty-day periods. SIRE database is small and is represented by a single volume.

The philosophy for spanning SOLIS database/volume data structures relies on the mapping of individual UDL files to one or more volumes of a given database. Therefore, there exists a single UDL file which corresponds to the single volume of the SIRE database; and there exists a set of UDL files corresponding to representative SOLIS PRODUCT volume combinations. All possible combinations of PRODUCT volumes generate a set much larger than the UDL file representation as currently employed. However, it is believed that many of these volume combinations are not particularly useful and that the following mappings are more than adequate in representing PRODUCT volumes:

| UDL File Name | SOLIS Database/Volume Indicators |
|---|---|
| SIRE | SIRE/R |
| PRODUCTS | PRODUCT/E and F (today's messages) |

| UDL File Name | SOLIS Database/Volume Indicators |
|---|---|
| PRODUCTA | PRODUCT/A, B, C, and D (current 15-day messages) |
| PRODUCTH | PRODUCT/H, G, J, K, L, M and N (7 days, non-edited messages) |
| PRODCUR1 | PRODUCT/B, A, C, D, E, F, G, H, J, and K (today's, current 15-day and non-edited 7-day messages) |
| PRODCUR2 | PRODUCT/C, A, B, D, E, F, G, H, J, and Z (Same as PROCUR1 plus the most current 60-day volume) |
| PRODOLD | PRODUCT/U, V, W, X, Y, Z, and T (all 60-day volumes) |
| PRODALL | PRODUCT/All available volumes |

The UDL file/SOLIS database/volume association is controlled by the tranfile definition of the UDL file. The tranfile header statement provides for the specification of a host specific filename for a given UDL file. For SOLIS files, this feature is exploited to specify the SOLIS database/volume association. This is a special use of this feature in the tranfile definition since it does not really map onto host recognizable names. This is because SOLIS does not have a true file concept as is common with the more general database management systems (e. g., the other three hosts accommodated by ADAPT I). Nonetheless, this feature in conjunction with some established conventions depended upon by the transformation software of ADAPT I provides a flexible means of associating various UDL file schemas with a given volume combination that may exist in SOLIS. These conventions are presented below.

## SOLIS Conventions

Currently, the ADAPT I TDL subsystem allows only up to ten characters for specifying a host specific filename. Keeping this limitation in mind, the transformation logic was set up to allow the following entries as host specific filenames. The single character '1' represents the SIRE database which only has one volume. The single character '2' represents all available volumes for the PRODUCT database, noting that available volumes on SOLIS fluctuate on any given day. The remaining volume combinations depend on the host specific filename which actually specifies the PRODUCT volumes (as presented above). For example, the UDL PRODUCTS file, maps onto today's volumes (which alternately fluctuates between volumes E and F) and has a host specific filename of EF. Similarly, UDL file PRODOLD maps onto all sixty-day volumes and has a host specific filename of UVWXYZT (i. e. , all letter designators for the available sixty-day volumes of PRODUCT). See figures D-1 and D-2. Finally, the first character of the host specific filename must be unique for each UDL file.

The general approach of this somewhat difficult transformation was to provide the user with a reasonable span of PRODUCT volumes that are normally used in every day operations. It is apparent from the above discussion that new volume combinations can be generated quite easily, if they are deemed necessary. It is important to note here that the seven UDL file mappings to seven different combinations of PRODUCT volumes all have the identical logical structure and hence the same schema (with the exception of the UDL filename). If one were not interested in retrieval efficiency one could entertain a single schema which mapped onto all available PRODUCT volumes (i. e. , the PRODALL file). The semantic meaning of this mapping would never be in doubt and the user would always be able to retrieve any message that he so desired. Of course, this approach is not practical since we are talking about hundreds of thousands of entries.

```
schema prodold remote(inter);
field windex mv char att(key,inv) 40 ;
field tildex mv char att(key, inv) 40 ;
field prodex sv char att(key, inv) 10 ;
field serdex sv char att(key, inv) 3 ;
field enddex sv char att(key, inv) 1 ;
field ddidex sv char att(key, inv) 3 ;
field disdex sv char att(key, inv) 10 ;
field isidex sv char att(key, inv) 3 ;
field tagdex sv char att(key, inv) 40 ;
field caveat sv char att(key, inv) 6 ;
field caveatcl sv char att(key, inv) 6 ;
field date sv num att(key, inv) 6 ;
field endate sv num att(key, inv) 6 ;
field dtgmin sv char att(key), inv) 2;
field dtghr sv char att(key, inv) 2;
field udn sv char att(key, inv) 10;
field followup sv char att(key, inv) 20 ;
field since sv char att(key, inv) 2;
field noshort sv char att(key, inv) 0 ;
field sectone sv char att(key, inv) 0 ;
field dailydis sv char att(key, inv) 0;
field ltitle sv char att(nkey,vis) 132 ;
field dtg sv char att(nkey, vis) 14 ;
field serial sv char att(key, vis) 20 ;
field tag sv char att(nkey, vis) 100 ;
field text sv char att(nkey, vis) v ;
eschema ;
```

Figure D-1. SOLIS Schema

```
tranfile prodold('UVWXYZT') database(solis) ;
field windex('; ') ;
field tildex('tildex ') ;
field prodex('prodex ') ;
field serdex('serdex ') ;
field enddex('enddex ') ;
field ddidex('ddidex ') ;
field disdex('disdex ') ;
field isidex('isidex ') ;
field tagdex('tagdex ') ;
field caveat('-') ;
field caveatcl('--') ;
field date ('d=') ;
field endate('ed=') ;
field dtgmin('dtgmin') ;
field dtghr('dtghr') ;
field udn('udn=') ;
field serial('s=', 'serial') ;
field followup('fs=') ;
field since('since') ;
field dailydis('dailydispatch') ;
field noshort('noshorttile') ;
field sectone('sectiononeonly') ;
field text('text') ;
field dtg('dtg') ;
field tag('tag') ;
field ltitle('title') ;
etran ;
```

Figure D-2. SOLIS Tranfile

## Date Data

Only date data elements in SOLIS can be classified as numeric. For PRODUCT, date data elements exist as DATE and ENDATE (see figure D-1). Both of these can be expressed in numerous ways in SOLIS, but for ADAPT I, it is mandatory to use year (two digits) followed by month (two digits) followed by day (two digits). This is because ADAPT I performs pure numeric validation against data constant entries involving the WRG and ORG operators, ensuring that the second operand is larger than the first. Therefore, one cannot express SOLIS date searches as numbers mixed with letters, the month before year, etc., as is possible in SOLIS.

## SOLIS Fields

The logical structure of a SOLIS database is generally simplistic and does not involve aggregate structures; i.e., UDL repeating groups or arrays. Therefore, all referenceable SOLIS data elements are mapped onto UDL fields. Some are termed as multivalued for semantic continuity, others are termed as single-valued. The distinction in this case is not important since the fields mapped as multivalued are not visible, only keyed; i.e., they can be queried but not displayed.

## PRODUCT Display

Currently, under ADAPT I, data elements which can either be displayed via SOLIS SS or A0 commands are legal in ADAPT I for display, (i.e., they are visible (VIS)). Therefore, for PRODUCT, only the complete message text, the formatted long title, serial number, date-time-group, and a tag list are displayable data elements. All others are given an invisible display attribute (INV). This convention is not random but is required for proper transformations. The majority of PRODUCT index terms are not retrievable for display via SOLIS; therefore, it is essentially impossible to display them as separate entities under ADAPT I.

## SIRE Display

This same uniform transform mechanism is used for the SIRE database, even though this database is structured such that all its indexable terms are formatted to such a degree that they could be selected as display items. In the future, the Target System Output Translator (TSOT) for the SOLIS host might be expanded to recognize these additional displayable data elements of SIRE.

## Host Specific Fieldnames

Another convention established for the SOLIS transformations and tranfile sequences involves the specification of host specific field names. For SOLIS index terms, such as windex, tildex and enddex, SOLIS expects a space between the index term name and the search-value. The space signifies the relational operation of equality. By convention, this space is not included in the decompiling tables (for the token EQ) but is instead included as part of the host specific fieldname as supplied in the tranfile sequence. For example, the UDL field TILDEX is represented as 'TILDEX ' in the TDL entry (i. e., field tildex ('tildex ');), note the trailing space. (See figure D-2).

Since equality is handled in the host specific field name entries via tranfile sequences, data elements such as caveatclass or followup, which do not have a space for equality, must not include the space in the host specific field name; i. e., field caveat ('-'); ... field followup ('fs=').

In some cases, hosts recognize different data element names for query and display requests. SOLIS is one such host. A data element such as serial has two host specific names associated with it: 's=' is the host specific name for query requests and 'serial' is the host specific name for display output (i. e., as displayed from A0 outputs).

## Summary

In summary, the following conventions must be abided by for creating schema/tranfile sequences for SOLIS files:

a. The tranfile host specific filename is used to specify SOLIS volume selection:

   1) '1' signifies SIRE.

   2) '2' signifies PRODUCT, all volumes.

   3) 'remaining values' signify the actual PRODUCT volumes to select (i. e., not suppress).

b. The first character of the tranfile host specific filename must be unique for all SOLIS UDL files.

c. The relational operation of equality is usually a space in SOLIS. However, this space is not handled by the decompiling tables and therefore must be included as the last character of the tranfile host specific fieldnames which require it.

d. Only SOLIS data elements (i. e., UDL fields) which accommodate all numeric relational operators can be given a data type of numeric (NUM). Currently, this is DATE and ENDATE of PRODUCT, and DATE of SIRE.

e. The only SOLIS data elements which can be given a display attribute of visible (VIS) are those which are retrievable by either the SS or A0 SOLIS commands. Therefore, only the complete message text (TEXT), the document long title (TITLE), the date-time-group (DTG), the serial (SERIAL), and a tag list (TAG) are designated as visible. A tag list is not retrievable from SIRE via A0.

f. The host specific field names specified in the tranfile sequence must match the field names as they exist on the A0 output; i. e., 'title', 'tag', 'serial' and 'dtg'. Two host specific field names are supplied for serial since its query name (the first, 'fs=') and display name ('serial') are different.

g. Since the basic document or message data is not known as a data element in SOLIS (i. e., does not have a name), the UDL field name of text is used to specify this data element.

h. Only true data elements which can be queried via the SOLIS free screen can be given the interrogation attribute of keyed (KEY). All others (which are visible) must be termed nonkeyed (NKEY).

i. SOLIS existence search terms (i. e., dailydispatch, sectiononeonly, and noshorttitle) must be given a zero field width in the schema sequence (see figure D-1).

j. Due to limitations on logical line lengths of the SOLIS free screen, keyed fields should not exceed a field width of 40 characters.

k. SOLIS files are always specified as REMOTE(INTER) in the schema header statement. (See figure D-1).

## SCHEMA/TRANFILE GENERATIONS FOR THE RYETIP HOST

### Data Recognition

The RYETIP host provides a general database management facility for its users; and, therefore, the transformation technology developed for this host is completely file independent. It does present, however, a somewhat unusual problem for the Target System Output Translation (TSOT) aspect of the transformation. In order to adequately and efficiently recognize RYETIP output from ADAPT I instigated queries, it was necessary to develop a data-by-position recognition mechanism. The data-by-position mechanism can be contrasted with the more conventional data-by-name mechanism as follows. The data-by-name mechanism requires that the data element name (host specific) be prefixed to its value in the record. Usually, this name/value association is separated by an equality sign; i.e., data-element-name = value. The data-by-position mechanism depends on the data element value existing in a specific position within the record. Therefore, it is important that the precise size of the record be known and that the position of a given data element value within a record be the same across all records for a given query (not necessarily across all queries). The data-by-position mechanism is used exclusively for all RYETIP transformations. These two mechanisms have been designed into the TDL subsystem, hence either can be utilized for a host, depending on which lends itself best to the transformation. All tranfile sequences for RYETIP files must utilize the data-by-position mechanism for data isolation.

## Data Structures

Currently, ADAPT I does not recognize CODASYL sets; and, consequently, RYETIP grouped multiformatted files must be treated as a group of identically structured UDL files. In order to force the RYETIP host to query only a single format, the host specific filename specified in the tranfile sequence must include the format name. The construct is 'host specific filename, host specific format name'. The record structure of RYETIP files is generally simple, not accommodating aggregate structures. However, both single and multivalued fields are present. All RYETIP fields are designated as single-valued unless they have a family name representing a TILE family group. The individual unit fields comprising the family field are mapped onto separate UDL single-valued fields (this is not a requirement; but, if the user desires to query one or more RYETIP unit fields individually, the mapping is needed).

## Numeric Data

Due to the strict UDL definition of a numeric data type, only those RYETIP fields which accommodate numeric data only are designated as numeric (NUM). This implies that all relational operators valid for UDL numeric fields are expressable in RYETIP.

## Character Data

Although RYETIP contains some fields which contain positional data, such as latitude or longitude data, the data type of those fields is termed character (CHAR), not geographic. This is because geographic fields behave in a precise manner in UDL and must comply to the three geographic relational operators described under UDL: ALONG, INSIDE and OUTSIDE. RYETIP operations against lat/long fields do not precisely comply to these operations.

## Attributes

All RYETIP fields can be queried and also displayed and, therefore, are given the interrogation and display attributes of keyed (KEY) and visible (VIS) respectively. The actual field widths specified in the schema sequences are obtainable from the RYETIP INFO verb output. Since the RYETIP transformations use the data-by-position mechanism, the precise field widths must be known; therefore, the correct position values must be used. Note the complete record size (as a number of characters) is derivable from the INFO verb output and must be supplied correctly in order for the transformations to perform properly. Again, the starting data position for each field, as well as the number of occurrences for the multivalued field definitions in the tranfile sequence, are derivable from the INFO verb output. The number of occurrences is the number of unit fields comprising the family.

## Summary

The above discussion for the RYETIP host is summarized as follows:

a. All RYETIP tranfile sequences utilize the data-by-position recognition mechanism.

b. Use the RYETIP INFO verb for extracting the appropriate positional information required for the tranfile sequence.

c. For RYETIP multiformatted files, repeat each format as an identically structured UDL file.

d. For multiformatted files, the tranfile host specific filename must include the desired host specific format name separated by a comma (i.e., 'host specific filename, host specific format name').

e. RYETIP family/unit data element sequences are mapped onto a multivalued field and a set of single-valued fields respectively.

f. RYETIP files are always specified in the schema header statement as REMOTE(BATCH). (See figure D-3 and D-4).

```
schema car remote(batch);
field make sv char att(key,vis) 25 ;
field base sv num att(key,vis) 6 ;
field long sv num att(key,vis) 6 ;
field horse sv num att(key,vis) 4 ;
field body sv char att(key,vis) 9 ;
field cube sv num att(key,vis) 3 ;
field belt sv char att(key,vis) 1 ;
field heat sv char att(key,vis) 1 ;
field gear sv char att(key,vis) 5 ;
field steer sv char att(key,vis) 5 ;
field pad sv char att(key,vis) 5 ;
field light sv char att(key,vis) 5 ;
field power sv char att(key,vis) 5 ;
field tire sv char att(key,vis) 5 ;
field seat sv char att(key,vis) 5 ;
field viewmirr sv char att(key,vis) 5 ;
field lid sv char att(key,vis) 5 ;
field trim sv char att(key,vis) 5 ;
field smoke sv char att(key,vis) 5 ;
field cons sv char att(key,vis) 5 ;
field glass sv char att(key,vis) 5 ;
field extra sv char att(key,vis) 5 ;
field rcost sv num att(key,vis) 4 ;
field fill sv char att(nkey,vis) 1 ;
field wcost sv num att(key,vis) 4 ;
eschema ;
```

Figure D-3. RYETIP Schema

```
tranfile car('car') database(ryetip) position(320) ;
field make pos(1) ;
field base pos(26) ;
field long pos(32) ;
field horse pos(38) ;
field body pos(42) ;
field cube pos(51) ;
field belt pos(54) ;
field heat pos(55) ;
field gear pos(56) ;
field steer pos(61) ;
field pad pos(66) ;
field light pos(71) ;
field power pos(76) ;
field tire pos(81) ;
field seat pos(86) ;
field viewmirr('view') pos(91) ;
field lid pos(96) ;
field trim pos(101) ;
field smoke pos(106) ;
field cons pos(111) ;
field glass pos(116) ;
field extra pos(121) ;
field rcost pos(126) ;
field fill pos(130) ;
field wcost pos(131) ;
etran ;
```

Figure D-4. RYETIP Tranfile

## SCHEMA/TRANFILE GENERATIONS FOR THE DIAOLS HOST

### General

The DIAOLS host is a general database management system, and therefore the ADAPT I transformation technology is completely file independent. The tranfile sequences utilize the data-by-name recognition mechanism and are completely free of any host dependent conventions. The transformations from UDL to DIAOLS, however, do operate over a set of rules which must be followed when constructing the appropriate schemas and tranfiles.

### Data Structures

Since a record-id data element is required for each DIAOLS file, its UDL name must be specified in the tranfile header statement as the argument for KEY. Currently, under ADAPT I, DIAOLS synonyms are not recognized. Therefore, only the original data elements in a DIAOLS file can be mapped onto by UDL. DIAOLS periodic elements are represented in UDL files as multivalued fields and simple data elements in DIAOLS are mapped onto single-valued fields. The special relational periodic element of DIAOLS is semantically equivalent to a single level repeating group in UDL. Each data element belonging to the relational periodic set is defined as a field of that repeating group definition.

### Attributes

Note, since the DIAOLS RELATE verb can only be utilized in a dependent sense in DIAOLS, all fields belonging to the UDL counterpart, repeating group, must be given an interrogation attribute of dependent (DEP). Although the DIAOLS RELATE operation must be initiated as a dependent operation the corresponding relational periodic elements are also periodic elements in DIAOLS and can be referenced in DIAOLS as independent entities (i.e., keyed). Therefore, a choice must be made for the schema/

tranfile sequences. If the UDL repeating group relationship, which must always be dependent, is not important with respect to the operational value of the file, then it is best to map the periodic relational elements onto individual keyed multivalued fields. They then can be referenced in independent UDL FIND statements (of course, they no longer can be associated as an occurrence group, or related in DIAOLS terminology).

Similarly, DIAOLS data element pairs (latitude/longitude) which map onto a UDL geographic data type must also be given an interrogation attribute of dependent. For this particular mapping, the ADAPT I transforms map the latitude and longitude data elements of DIAOLS onto a single UDL lat/long positional field. This field is given a different name in UDL and requires two host specific field names, one for the latitude which must be supplied first, and one for the longitude (see field 'ppoint' in figures D-5, D-6). For brevity in query generation and query response output, all tranfile sequences utilize the DIAOLS element-short-name for host specific field names. Note, the UDL repeating group name is completely arbitrary, and does not have a host specific counterpart (i.e., they do not exist as such in DIAOLS).

All DIAOLS data elements are displayable, hence a display attribute of visible (VIS) is assigned to all UDL fields in the schema.

## Summary

The following summarizes the rules and conventions for the generation of schema/tranfile sequences for the DIAOLS host:

a. All fields defined for a repeating group must be given the interrogation attribute of dependent (DEP).

b. All fields assigned as geographic must also be assigned the interrogation attribute of dependent.

```
schema guide remote(batch) ;
field datechng sv char att(key,vis) 8 ;
field recordid sv char att(key,vis) 4 ;
field name sv char att(key,vis) 20 ;
field latitude sv char att(key,vis) 7 ;
field longtude sv char att(key,vis) 8 ;
field ppoint sv geo att(dep,vis) 22 ;
field category sv num att(key,vis) 5 ;
field country sv char att(key,vis) 2 ;
field status sv char att(key,vis) 10 ;
field usage sv char att(key,vis) 10 ;
field navships sv char att(key,vis) 16 ;
field location sv char att(key,vis) 14 ;
field remarks1 sv char att(key,vis) 20 ;
field remarks2 sv char att(key,vis) 20 ;
rgroup msndata ;
field msndate sv num att(dep,vis) 8 ;
field msnumber mv char att(dep,vis) 10 ;
ergroup ;
eschema ;
```

Figure D-5. DIAOLS Schema

```
tranfile guide database(diaols) key(recordid) ;
field datechng('0001') ;
field recordid('0002') ;
field name('0003') ;
field latitude('0004') ;
field longtude('0005') ;
field category('0006') ;
field country('0007') ;
field status('0008') ;
field usage('0009') ;
field navships('0012') ;
field location('0013') ;
field remarks1('0014') ;
field remarks2('0015') ;
field msndate('0010') ;
field msnumber('0011') ;
field ppoint('0004','0005') ;
etran ;
```

Figure D-6. DIAOLS Tranfile

c. All tranfile sequences must supply the element-short-names of both the DIAOLS latitude data element and the longitude data element, (in that order) for UDL geographic fields.

d. UDL fields are mapped onto original DIAOLS data elements. Do not use synonym mappings.

e. The DIAOLS record-id data element must always be specified in the tranfile header statement as the key field (KEY argument).

f. Until the UDL 'contains' relational operator is implemented, it is best to designate large remarks type data elements as non-keyed (NKEY).

g. DIAOLS files are always specified as REMOTE(BATCH) in the schema header statement (see figure D-5).

## SCHEMA/TRANFILE GENERATIONS FOR THE ISSPIC HOST

### General

As was the case for RYETIP and DIAOLS, the DMS-1100/QLP system at ISSPIC provides users with a general database management system. Therefore, the conventions utilized for the schema and tranfile sequences are formal and independent of files. The designated data structure mappings from UDL to data structures of the DMS-1100/QLP database management system (currently in the process of being installed at ISSPIC) are precise and must be followed exactly at all times.

### Data Structures

Data structure mappings are very straightforward for UDL/QLP transformations. It is important that a key field be specified on all QLP files. This designated field is used by the Target System Output Translator to determine the beginning of a new record. This key field is the record-id of the QLP record-type.

It is also important to note that individual record-types comprising an area in QLP are mapped onto unique UDL files. QLP set relations are not recognized under the current ADAPT I. Sets will be installed in UDL in a later phase. The host specific filename used in the tranfile sequence must be the name of the record-type mapped onto in QLP. In order to eliminate any possible ambiguity, the host specific filename is used to qualify all field references. This is because QLP allows interrogation statements to span more than one record-type. The qualifying record-type must be specified if the field name happens to exist in more than one record-type in the area.

All fields are keyed and visible. As with the other systems, only those fields which contain pure numeric data can be given the data attribute of numeric (NUM). The UDL array aggregate structure maps directly onto the DMS-1100 COBOL table or array data structure. It is necessary to specify all host specific aggregate names (i. e., names for UDL arrays) in the tranfile sequence since they can be referenced in DISPLAY statements individually or as arguments of TREE, (i. e., the transformations utilize this name for outputting an entire array).

## Summary

In summary, the following set of rules and/or conventions must be followed rigorously in generating ISSPIC (DMS-1100/QLP) schema/tranfile sequences:

a. The host specific filename in the tranfile sequence must specify the record-type name.

b. All fields in a given UDL file must be contained in an individual record-type. That is, spanning multiple record-types with a single UDL file is prohibited.

c. All aggregates (arrays) must have their host specific names supplied in the tranfile sequence.

d. A key field must be supplied in the tranfile header statement. This field must be the record-id data element of the QLP file.

e. DMS-1100/QLP files are always specified as REMOTE(INTER) in the schema header statement (see figure D-7 and D-8).

```
schema employee remote(inter) ;
field lastname sv char att(key,vis) 10   ;
field firstnam sv char att(key,vis) 10   ;
field title sv char att(key,vis) 12   ;
field salary sv num att(key,vis) 5    ;
field spousnam sv char att(key,vis) 10   ;
array children 16       ;
field childnam sv char att(key,vis) 10   ;
field hobbies mv char att(key,vis) 10   ;
earray;
array expernce 10  ;
field company sv char att(key,vis) 15   ;
field numyears sv num att(key,vis) 2    ;
earray;
array projects 5 expernce  ;
field super sv char att(key,vis) 25  ;
field ptitle sv char att(key,vis) 12  ;
earray ;
eschema  ;
```

Figure D-7. ISSPIC Schema

```
tranfile employee('person-f') database(isspic) key(lastname) ;
field lastname('name-l') ;
field firstnam('name-f') ;
field title('position') ;
field salary('wages') ;
field spousnam('name-w-h');
field childnam('name-c') ;
field hobbies('hob') ;
field company('corp') ;
field numyears('years') ;
field super('supervisor') ;
field ptitle('proj-title') ;
array children('child') ;
array expernce('experience') ;
array projects('proj') ;
etran;
```

Figure D-8. ISSPIC Tranfile

## DISTRIBUTION LIST

| | |
|---|---|
| Defense Documentation Center<br>Cameron Station<br>Alexandria, VA 22314 | 12 copies |
| Office of Naval Research<br>Information Systems Program<br>Code 437<br>Arlington, VA 22217 | 2 copies |
| Office of Naval Research<br>Code 102IP<br>Arlington, VA 22217 | 6 copies |
| Office of Naval Research<br>Code 200<br>Arlington, VA 22217 | 1 copy |
| Office of Naval Research<br>Code 455<br>Arlington, VA 22217 | 1 copy |
| Office of Naval Research<br>Code 458<br>Arlington, VA 22217 | 1 copy |
| Office of Naval Research<br>Branch Office, Boston<br>495 Summer Street<br>Boston, MA 02210 | 1 copy |
| Office of Naval Research<br>Branch Office, Chicago<br>536 South Clark Street<br>Chicago, IL 60605 | 1 copy |
| Office of Naval Research<br>Branch Office, Pasadena<br>1030 East Green Street<br>Pasadena, CA 91106 | 1 copy |

|  |  |
|---|---|
| New York Area Office<br>715 Broadway — 5th Floor<br>New York, NY 10003 | 1 copy |
| Naval Research Laboratory<br>Technical Information Division, Code 2627<br>Washington, D. C. 20375 | 6 copies |
| Dr. A. L. Slafkosky<br>Scientific Advisor<br>Commandant of the Marine Corps (Code RD-1)<br>Washington, D. C. 20380 | 1 copy |
| Naval Electronics Laboratory Center<br>Advanced Software Technology Division<br>Code 5200<br>San Diego, CA 92152 | 1 copy |
| Mr. E. H. Gleissner<br>Naval Ship Research & Development Center<br>Computation and Mathematics Department<br>Bethesda, MD 20084 | 1 copy |
| Captain Grace M. Hopper<br>NAICOM/MIS Planning Branch (OP-916D)<br>Office of Chief of Naval Operations<br>Washington, D. C. 20350 | 1 copy |
| Mr. Kin B. Thompson<br>Technical Director<br>Information Systems Division (OP-91T)<br>Office of Chief of Naval Operations<br>Washington, D. C. 20305 | 1 copy |
| Advanced Research Projects Agency<br>Information Processing Techniques<br>1400 Wilson Boulevard<br>Arlington, VA 22209 | 1 copy |